The Julia Language

V1.13.0-rc4

The Julia Project

September 3, 2026

Contents

Part I

Manual

Chapter 1

Julia 1.13-rc4 Documentation

Welcome to the documentation for Julia 1.13-rc4.

Work in progress!

This documentation is for an unreleased, in-development, version of Julia.

Please read the release notes to see what has changed since the last release.

Note

The documentation is also available in PDF format: julia-1.13.0-rc4.pdf.

1.1 Introduction

Scientific computing has traditionally required the highest performance, yet domain experts have largely moved to slower dynamic languages for daily work. We believe there are many good reasons to prefer dynamic languages for these applications, and we do not expect their use to diminish. Fortunately, modern language design and compiler techniques make it possible to mostly eliminate the performance trade-off and provide a single environment productive enough for prototyping and efficient enough for deploying performance-intensive applications. The Julia programming language fills this role: it is a flexible dynamic language, appropriate for scientific and numerical computing, with performance comparable to traditional statically-typed languages.

Because Julia's compiler is different from the interpreters used for languages like Python or R, you may find that Julia's performance is unintuitive at first. If you find that something is slow, we highly recommend reading through the Performance Tips section before trying anything else. Once you understand how Julia works, it is easy to write code that is nearly as fast as C.

1.2 Julia Compared to Other Languages

Julia features optional typing, multiple dispatch, and good performance, achieved using type inference and just-in-time (JIT) compilation (and optional ahead-of-time compilation), implemented using LLVM. It is multi-paradigm, combining features of imperative, functional, and object-oriented programming. Julia provides ease and expressiveness for high-level numerical computing, in the same way as languages such as R, MATLAB, and Python, but also supports general programming. To achieve this, Julia builds upon the

lineage of mathematical programming languages, but also borrows much from popular dynamic languages, including Lisp, Perl, Python, Lua, and Ruby.

The most significant departures of Julia from typical dynamic languages are:

- The core language imposes very little; Julia Base and the standard library are written in Julia itself, including primitive operations like integer arithmetic
- A rich language of types for constructing and describing objects, that can also optionally be used to make type declarations
- The ability to define function behavior across many combinations of argument types via multiple dispatch
- Automatic generation of efficient, specialized code for different argument types
- Good performance, approaching that of statically-compiled languages like C

Although one sometimes speaks of dynamic languages as being "typeless", they are definitely not. Every object, whether primitive or user-defined, has a type. The lack of type declarations in most dynamic languages, however, means that one cannot instruct the compiler about the types of values, and often cannot explicitly talk about types at all. In static languages, on the other hand, while one can – and usually must – annotate types for the compiler, types exist only at compile time and cannot be manipulated or expressed at run time. In Julia, types are themselves run-time objects, and can also be used to convey information to the compiler.

What Makes Julia, Julia?

While the casual programmer need not explicitly use types or multiple dispatch, they are the core unifying features of Julia: functions are defined on different combinations of argument types, and applied by dispatching to the most specific matching definition. This model is a good fit for mathematical programming, where it is unnatural for the first argument to "own" an operation as in traditional object-oriented dispatch. Operators are just functions with special notation – to extend addition to new user-defined data types, you define new methods for the + function. Existing code then seamlessly applies to the new data types.

Partly because of run-time type inference (augmented by optional type annotations), and partly because of a strong focus on performance from the inception of the project, Julia's computational efficiency exceeds that of other dynamic languages, and even rivals that of statically-compiled languages. For large scale numerical problems, speed always has been, continues to be, and probably always will be crucial: the amount of data being processed has easily kept pace with Moore's Law over the past decades.

Advantages of Julia

Julia aims to create an unprecedented combination of ease-of-use, power, and efficiency in a single language. In addition to the above, some advantages of Julia over comparable systems include:

- Free and open source (MIT licensed)
- User-defined types are as fast and compact as built-ins
- No need to vectorize code for performance; devectorized code is fast

- Designed for parallelism and distributed computation
- Lightweight "green" threading (coroutines)
- Unobtrusive yet powerful type system
- Elegant and extensible conversions and promotions for numeric and other types
- Efficient support for Unicode, including but not limited to UTF-8
- Call C functions directly (no wrappers or special APIs needed)
- Powerful shell-like capabilities for managing other processes
- Lisp-like macros and other metaprogramming facilities

1.3 Important Links

A non-exhaustive list of links that will be useful as you learn and use the Julia programming language:

- Julia Homepage
- Install Julia
- Discussion forum
- Julia YouTube
- Find Julia Packages
- Learning Resources

Chapter 2

Getting Started

Julia installation is straightforward, whether using precompiled binaries or compiling from source. Download and install Julia by following the instructions at https://julialang.org/install/.

If you are coming to Julia from one of the following languages, then you should start by reading the section on noteworthy differences from MATLAB, R, Python, C/C++ or Common Lisp. This will help you avoid some common pitfalls since Julia differs from those languages in many subtle ways.

The easiest way to learn and experiment with Julia is by starting an interactive session (also known as a read-eval-print loop or "REPL") by double-clicking the Julia executable or running `julia` from the command line:

```
$ julia


               _
   _       _ _(_)_     |  Documentation: https://docs.julialang.org
  (_)     | (_) (_)    |
   _ _   _| |_  __ _   |  Type "?" for help, "]?" for Pkg help.
  | | | | | | |/ _` |  |
  | | |_| | | | (_| |  |  Version 1.13.0-rc4 (2026-09-02)
 _/ |\__'_|_|_|\__'_|  |  Official https://julialang.org release
|__/                   |


julia> 1 + 2
3

julia> ans
3
```

To exit the interactive session, type `CTRL-D` (press the Control/^ key together with the `d` key), or type `exit()`. When run in interactive mode, `julia` displays a banner and prompts the user for input. Once the user has entered a complete expression, such as `1 + 2`, and hits enter, the interactive session evaluates the expression and shows its value. If an expression is entered into an interactive session with a trailing semicolon, its value is not shown. The variable `ans` is bound to the value of the last evaluated expression whether it is shown or not. The `ans` variable is only bound in interactive sessions, not when Julia code is run in other ways.

To evaluate expressions written in a source file `file.jl`, write `include("file.jl")`.

To run code in a file non-interactively, you can give it as the first argument to the `julia` command:

```
$ julia script.jl
```

You can pass additional arguments to Julia, and to your program `script.jl`. A detailed list of all the available options can be found under Command-line Interface.

2.1 Resources

A curated list of useful learning resources to help new users get started can be found on the learning page of the main Julia website.

You can use the REPL as a learning resource by switching into the help mode. Switch to help mode by pressing `?` at an empty `julia>` prompt, before typing anything else. Typing a keyword in help mode will fetch the documentation for it, along with examples. Similarly for most functions or other objects you might encounter!

```
help?> begin
search: begin disable_sigint reenable_sigint

  begin

  begin...end denotes a block of code.
```

If you already know Julia a bit, you might want to peek ahead at Performance Tips and Workflow Tips, or check out the comprehensive ModernJuliaWorkflows blog.

Chapter 3

Installation

There are many ways to install Julia. The following sections highlight the recommended method for each of the main supported platforms, and then present alternative ways that might be useful in specialized situations.

The current installation recommendation is a solution based on Juliaup. If you installed Julia previously with a method that is *not* based on Juliaup and want to switch your system to an installation that is based on Juliaup, we recommend that you uninstall all previous Julia versions, ensure that you remove anything Julia related from your PATH variable and then install Julia with one of the methods described below.

3.1 Windows

On Windows Julia can be installed directly from the Windows store here. One can also install exactly the same version by executing

```
winget install --name Julia --id 9NJNWW8PVKMN -e -s msstore
```

in any shell.

3.2 Mac and Linux

Julia can be installed on Linux or Mac by executing

```
curl -fsSL https://install.julialang.org | sh
```

in a shell.

Command line arguments

One can pass various command line arguments to the Julia installer. The syntax for installer arguments is

```
curl -fsSL https://install.julialang.org | sh -s -- <ARGS>
```

Here <ARGS> should be replaced with one or more of the following arguments:

- `--yes` (or `-y`): Run the installer in a non-interactive mode. All configuration values use their default or a value supplied as a command line argument.
- `--default-channel=<NAME>`: Configure the default Juliaup channel. For example `--default-channel lts` would install the `lts` channel and configure it as the default.
- `--add-to-path=<yes|no>`: Configure whether Julia should be added to the PATH environment variable. Valid values are `yes` (default) and `no`.
- `--background-selfupdate=<SECONDS>`: Configure an optional CRON job that auto-updates Juliaup if `<SECONDS>` has a value larger than 0. The actual value controls how often the CRON job will run to check for a new Juliaup version in seconds. The default value is 0, i.e. no CRON job will be created.
- `--startup-selfupdate=<MINUTES>`: Configure how often Julia will check for new versions of Juliaup when Julia is started. The default is every 1440 minutes.
- `-p=<PATH>` (or `--path`): Configure where the Julia and Juliaup binaries are installed. The default is `~/.juliaup`.

3.3 Alternative installation methods

Note that we recommend the following methods *only* if none of the installation methods described above work for your system.

Some of the installation methods described below recommend installing a package called `juliaup`. Note that this nevertheless installs a fully functional Julia system, not just Juliaup.

App Installer (Windows)

If the Windows Store is blocked on a system, we have an alternative MSIX App Installer based setup. To use the App Installer version, download this file and open it by double clicking on it. One can also install exactly the same version by executing the PowerShell command

```
Add-AppxPackage -AppInstallerFile https://install.julialang.org/Julia.appinstaller
```

MSI Installer (Windows)

If neither the Windows Store nor the App Installer version work on your Windows system, you can also use a MSI based installer. Note that this installation methods comes with serious limitations and is generally not recommended unless no other method works. For example, there is no automatic update mechanism for Juliaup with this installation method. The 64 bit version of the MSI installer can be downloaded from here and the 32 bit version from here.

By default the install will be a per-user install that does not require elevation. You can also do a system install by running the following command from a shell:

```
msiexec /i <PATH_TO_JULIA_MSI> ALLUSERS=1
```

Homebrew (Mac and Linux)

On systems with brew, you can install Julia by running

```
brew install juliaup
```

in a shell. Note that you will have to update Juliaup with standard brew commands.

Arch Linux - AUR (Linux)

On Arch Linux, Juliaup is available in the Arch User Repository (AUR).

openSUSE Tumbleweed (Linux)

On openSUSE Tumbleweed, you can install Julia by running

```
zypper install juliaup
```

in a shell with root privileges.

cargo (Windows, Mac and Linux)

To install Julia via Rust's cargo, run:

```
cargo install juliaup
```

Chapter 4

Variables

A variable, in Julia, is a name associated (or bound) to a value. It's useful when you want to store a value (that you obtained after some math, for example) for later use. For example:

```
julia> x = 10   # Assign the value 10 to the variable x
10

julia> x + 1    # Doing math with x's value
11

julia> x = 1 + 1   # Reassign x's value
2

julia> x = "Hello World!"   # You can assign values of other types, like strings of text
"Hello World!"
```

Julia provides an extremely flexible system for naming variables. Variable names are case-sensitive, and have no semantic meaning (that is, the language will not treat variables differently based on their names).

```
julia> x = 1.0
1.0

julia> y = -3
-3

julia> Z = "My string"
"My string"

julia> customary_phrase = "Hello world!"
"Hello world!"

julia> UniversalDeclarationOfHumanRightsStart = "人人生而自由，在尊严和权利上一律平等。"
"人人生而自由，在尊严和权利上一律平等。"
```

Unicode names (in UTF-8 encoding) are allowed:

```
julia> δ = 0.00001
1.0e-5

julia> □□□□□ = "Hello"
"Hello"
```

In the Julia REPL and several other Julia editing environments, you can type many Unicode math symbols by typing the backslashed LaTeX symbol name followed by tab. For example, the variable name δ can be entered by typing `\delta`-*tab*, or even $\alpha^{(2)}$ by `\alpha`-*tab*-`\hat`-*tab*-`\^(2)`-*tab*. (If you find a symbol somewhere, e.g. in someone else's code, that you don't know how to type, the REPL help will tell you: just type ? and then paste the symbol.)

Julia will even let you shadow existing exported constants and functions with local ones (although this is not recommended to avoid potential confusions):

```
julia> pi = 3
3

julia> pi
3

julia> sqrt = 4
4

julia> length() = 5
length (generic function with 1 method)

julia> Base.length
length (generic function with 79 methods)
```

However, if you try to redefine a built-in constant or function that you have explicitly imported, Julia will give you an error:

```
julia> using Base: pi, sqrt

julia> pi
π = 3.1415926535897...

julia> pi = 3
ERROR: cannot assign a value to imported variable Base.pi from module Main

julia> sqrt(100)
10.0

julia> sqrt = 4
ERROR: cannot assign a value to imported variable Base.sqrt from module Main
```

Julia 1.12

Note that in versions prior to Julia 1.12, these errors depended on *use* rather than definition of the conflicting binding.

4.1 Allowed Variable Names

Variable names must begin with a letter (A-Z or a-z), underscore, or a subset of Unicode code points greater than 00A0; in particular, Unicode character categories Lu/Ll/Lt/Lm/Lo/Nl (letters), Sc/So (currency and other symbols), and a few other letter-like characters (e.g. a subset of the Sm math symbols) are allowed. Subsequent characters may also include ! and digits (0-9 and other characters in categories Nd/No), as well as other Unicode code points: diacritics and other modifying marks (categories Mn/Mc/Me/Sk), some punctuation connectors (category Pc), primes, and a few other characters.

Operators like + are also valid identifiers, but are parsed specially. In some contexts, operators can be used just like variables; for example `(+)` refers to the addition function, and `(+) = f` will reassign it. Most of the Unicode infix operators (in category Sm), such as ⊕, are parsed as infix operators and are available for user-defined methods (e.g. you can use `const ⊗ = kron` to define ⊗ as an infix Kronecker product). Operators can also be suffixed with modifying marks, primes, and sub/superscripts, e.g. +̂ₐ″ is parsed as an infix operator with the same precedence as +. A space is required between an operator that ends with a subscript/superscript letter and a subsequent variable name. For example, if +ᵃ is an operator, then +ᵃx must be written as +ᵃ x to distinguish it from + ᵃx where ᵃx is the variable name.

A particular class of variable names is one that contains only underscores. These identifiers are write-only. I.e. they can only be assigned values, which are immediately discarded, and their values cannot be used in any way.

```
julia> x, ___ = size([2 2; 1 1])
(2, 2)

julia> y = ___
ERROR: syntax: all-underscore identifiers are write-only and their values cannot be used in
↪  expressions

julia> println(___)
ERROR: syntax: all-underscore identifiers are write-only and their values cannot be used in
↪  expressions
```

The only explicitly disallowed names for variables are the names of the built-in Keywords:

```
julia> else = false
ERROR: ParseError:
# Error @ none:1:1
else = false
└──┘ ── invalid identifier

julia> try = "No"
ERROR: ParseError:
# Error @ none:1:1
try = "No"
└─────┘ ── try without catch or finally
```

Some Unicode characters are considered to be equivalent in identifiers. Different ways of entering Unicode combining characters (e.g., accents) are treated as equivalent (specifically, Julia identifiers are NFC. Julia also includes a few non-standard equivalences for characters that are visually similar and are easily entered by some input methods. The Unicode characters ɛ (U+025B: Latin small letter open e) and µ (U+00B5:

micro sign) are treated as equivalent to the corresponding Greek letters. The middle dot · (U+00B7) and the Greek interpunct · (U+0387) are both treated as the mathematical dot operator · (U+22C5). The minus sign − (U+2212) is treated as equivalent to the hyphen-minus sign - (U+002D).

4.2 Assignment expressions and assignment versus mutation

An assignment `variable = value` "binds" the name `variable` to the `value` computed on the right-hand side, and the whole assignment is treated by Julia as an expression equal to the right-hand-side `value`. This means that assignments can be *chained* (the same `value` assigned to multiple variables with `variable1 = variable2 = value`) or used in other expressions, and is also why their result is shown in the REPL as the value of the right-hand side. (In general, the REPL displays the value of whatever expression you evaluate.) For example, here the value 4 of `b = 2+2` is used in another arithmetic operation and assignment:

```
julia> a = (b = 2+2) + 3
7

julia> a
7

julia> b
4
```

A common confusion is the distinction between *assignment* (giving a new "name" to a value) and *mutation* (changing a value). If you run `a = 2` followed by `a = 3`, you have changed the "name" a to refer to a new value 3 ... you haven't changed the number 2, so 2+2 will still give 4 and not 6! This distinction becomes more clear when dealing with *mutable* types like arrays, whose contents *can* be changed:

```
julia> a = [1,2,3] # an array of 3 integers
3-element Vector{Int64}:
 1
 2
 3

julia> b = a   # both b and a are names for the same array!
3-element Vector{Int64}:
 1
 2
 3
```

Here, the line `b = a` does *not* make a copy of the array a, it simply binds the name b to the *same* array a: both b and a "point" to one array `[1,2,3]` in memory. In contrast, an assignment `a[i] = value` *changes* the *contents* of the array, and the modified array will be visible through both the names a and b:

```
julia> a[1] = 42     # change the first element
42

julia> a = 3.14159   # a is now the name of a different object
3.14159
```

```
julia> b   # b refers to the original array object, which has been mutated
3-element Vector{Int64}:
 42
  2
  3
```

That is, `a[i] = value` (an alias for `setindex!`) *mutates* an existing array object in memory, accessible via either `a` or `b`. Subsequently setting `a = 3.14159` does not change this array, it simply binds `a` to a different object; the array is still accessible via `b`. Another common syntax to mutate an existing object is `a.field = value` (an alias for `setproperty!`), which can be used to change a `mutable struct`. There is also mutation via dot assignment, for example `b .= 5:7` (which mutates our array `b` in-place to contain `[5,6,7]`), as part of Julia's vectorized "dot" syntax.

When you call a function in Julia, it behaves as if you *assigned* the argument values to new variable names corresponding to the function arguments, as discussed in Argument-Passing Behavior. (By convention, functions that mutate one or more of their arguments have names ending with `!`.)

4.3 Stylistic Conventions

While Julia imposes few restrictions on valid names, it has become useful to adopt the following conventions:

- Names of variables are in lower case.
- Word separation can be indicated by underscores (`'_'`), but use of underscores is discouraged unless the name would be hard to read otherwise.
- Names of Types and `Modules` begin with a capital letter and word separation is shown with upper camel case instead of underscores.
- Names of `functions` and `macros` are in lower case, without underscores.
- Functions that write to their arguments have names that end in `!`. These are sometimes called "mutating" or "in-place" functions because they are intended to produce changes in their arguments after the function is called, not just return a value.

For more information about stylistic conventions, see the Style Guide.

Chapter 5

Integers and Floating-Point Numbers

Integers and floating-point values are the basic building blocks of arithmetic and computation. Built-in representations of such values are called numeric primitives, while representations of integers and floating-point numbers as immediate values in code are known as numeric literals. For example, `1` is an integer literal, while `1.0` is a floating-point literal; their binary in-memory representations as objects are numeric primitives.

Julia provides a broad range of primitive numeric types, and a full complement of arithmetic and bitwise operators as well as standard mathematical functions are defined over them. These map directly onto numeric types and operations that are natively supported on modern computers, thus allowing Julia to take full advantage of computational resources. Additionally, Julia provides software support for Arbitrary Precision Arithmetic, which can handle operations on numeric values that cannot be represented effectively in native hardware representations, but at the cost of relatively slower performance.

The following are Julia's primitive numeric types:

- **Integer types:**

Type	Signed?	Number of bits	Smallest value	Largest value
`Int8`	✓	8	-2^7	2^7 - 1
`UInt8`		8	0	2^8 - 1
`Int16`	✓	16	-2^15	2^15 - 1
`UInt16`		16	0	2^16 - 1
`Int32`	✓	32	-2^31	2^31 - 1
`UInt32`		32	0	2^32 - 1
`Int64`	✓	64	-2^63	2^63 - 1
`UInt64`		64	0	2^64 - 1
`Int128`	✓	128	-2^127	2^127 - 1
`UInt128`		128	0	2^128 - 1
`Bool`	N/A	8	`false` (0)	`true` (1)

- **Floating-point types:**

Type	Precision	Number of bits
`Float16`	half	16
`Float32`	single	32
`Float64`	double	64

Additionally, full support for Complex and Rational Numbers is built on top of these primitive numeric types. All numeric types interoperate naturally without explicit casting, thanks to a flexible, user-extensible type promotion system.

5.1 Integers

Literal integers are represented in the standard manner:

```
julia> 1
1

julia> 1234
1234
```

The default type for an integer literal depends on whether the target system has a 32-bit architecture or a 64-bit architecture:

```
# 32-bit system:
julia> typeof(1)
Int32

# 64-bit system:
julia> typeof(1)
Int64
```

The Julia internal variable `Sys.WORD_SIZE` indicates whether the target system is 32-bit or 64-bit:

```
# 32-bit system:
julia> Sys.WORD_SIZE
32

# 64-bit system:
julia> Sys.WORD_SIZE
64
```

Julia also defines the types `Int` and `UInt`, which are aliases for the system's signed and unsigned native integer types respectively:

```
# 32-bit system:
julia> Int
Int32
julia> UInt
```

```
UInt32

# 64-bit system:
julia> Int
Int64
julia> UInt
UInt64
```

Larger integer literals that cannot be represented using only 32 bits but can be represented in 64 bits always create 64-bit integers, regardless of the system type:

```
# 32-bit or 64-bit system:
julia> typeof(3000000000)
Int64
```

Unsigned integers are input and output using the `0x` prefix and hexadecimal (base 16) digits `0-9a-f` (the capitalized digits `A-F` also work for input). The size of the unsigned value is determined by the number of hex digits used:

```
julia> x = 0x1
0x01

julia> typeof(x)
UInt8

julia> x = 0x123
0x0123

julia> typeof(x)
UInt16

julia> x = 0x1234567
0x01234567

julia> typeof(x)
UInt32

julia> x = 0x123456789abcdef
0x0123456789abcdef

julia> typeof(x)
UInt64

julia> x = 0x11112222333344445555666677778888
0x11112222333344445555666677778888

julia> typeof(x)
UInt128
```

This behavior is based on the observation that when one uses unsigned hex literals for integer values, one typically is using them to represent a fixed numeric byte sequence, rather than just an integer value.

Binary and octal literals are also supported:

```
julia> x = 0b10
0x02

julia> typeof(x)
UInt8

julia> x = 0o010
0x08

julia> typeof(x)
UInt8

julia> x = 0x00000000000000001111222233334444
0x00000000000000001111222233334444

julia> typeof(x)
UInt128
```

As for hexadecimal literals, binary and octal literals produce unsigned integer types. The size of the binary data item is the minimal needed size, if the leading digit of the literal is not `0`. In the case of leading zeros, the size is determined by the minimal needed size for a literal, which has the same length but leading digit 1. It means that:

- `0x1` and `0x12` are `UInt8` literals,
- `0x123` and `0x1234` are `UInt16` literals,
- `0x12345` and `0x12345678` are `UInt32` literals,
- `0x123456789` and `0x1234567890adcdef` are `UInt64` literals, etc.

Even if there are leading zero digits which don't contribute to the value, they count for determining storage size of a literal. So `0x01` is a `UInt8` while `0x0001` is a `UInt16`.

That allows the user to control the size.

Unsigned literals (starting with `0x`) that encode integers too large to be represented as `UInt128` values will construct `BigInt` values instead. This is not an unsigned type but it is the only built-in type big enough to represent such large integer values.

Binary, octal, and hexadecimal literals may be signed by a `-` immediately preceding the unsigned literal. They produce an unsigned integer of the same size as the unsigned literal would do, with the two's complement of the value:

```
julia> -0x2
0xfe

julia> -0x0002
0xfffe
```

The minimum and maximum representable values of primitive numeric types such as integers are given by the `typemin` and `typemax` functions:

```
julia> (typemin(Int32), typemax(Int32))
(-2147483648, 2147483647)

julia> for T in [Int8,Int16,Int32,Int64,Int128,UInt8,UInt16,UInt32,UInt64,UInt128]
           println("$(lpad(T,7)): [$(typemin(T)),$(typemax(T))]")
       end
   Int8: [-128,127]
  Int16: [-32768,32767]
  Int32: [-2147483648,2147483647]
  Int64: [-9223372036854775808,9223372036854775807]
 Int128: [-170141183460469231731687303715884105728,170141183460469231731687303715884105727]
  UInt8: [0,255]
 UInt16: [0,65535]
 UInt32: [0,4294967295]
 UInt64: [0,18446744073709551615]
UInt128: [0,340282366920938463463374607431768211455]
```

The values returned by `typemin` and `typemax` are always of the given argument type. (The above expression uses several features that have yet to be introduced, including for loops, Strings, and Interpolation, but should be easy enough to understand for users with some existing programming experience.)

Overflow behavior

In Julia, exceeding the maximum representable value of a given type results in a wraparound behavior:

```
julia> x = typemax(Int64)
9223372036854775807

julia> x + 1
-9223372036854775808

julia> x + 1 == typemin(Int64)
true
```

Arithmetic operations with Julia's integer types inherently perform modular arithmetic, mirroring the characteristics of integer arithmetic on modern computer hardware. In scenarios where overflow is a possibility, it is crucial to explicitly check for wraparound effects that can result from such overflows. The `Base.Checked` module provides a suite of arithmetic operations equipped with overflow checks, which trigger errors if an overflow occurs. For use cases where overflow cannot be tolerated under any circumstances, utilizing the `BigInt` type, as detailed in Arbitrary Precision Arithmetic, is advisable.

An example of overflow behavior and how to potentially resolve it is as follows:

```
julia> 10^19
-8446744073709551616

julia> big(10)^19
10000000000000000000
```

Division errors

Integer division (the `div` function) has two exceptional cases: dividing by zero, and dividing the lowest negative number (`typemin`) by -1. Both of these cases throw a `DivideError`. The remainder and modulus functions (`rem` and `mod`) throw a `DivideError` when their second argument is zero.

5.2 Floating-Point Numbers

Literal floating-point numbers are represented in the standard formats, using E-notation when necessary:

```
julia> 1.0
1.0

julia> 1.
1.0

julia> 0.5
0.5

julia> .5
0.5

julia> -1.23
-1.23

julia> 1e10
1.0e10

julia> 2.5e-4
0.00025
```

The above results are all `Float64` values. Literal `Float32` values can be entered by writing an `f` in place of e:

```
julia> x = 0.5f0
0.5f0

julia> typeof(x)
Float32

julia> 2.5f-4
0.00025f0
```

Values can be converted to `Float32` easily:

```
julia> x = Float32(-1.5)
-1.5f0

julia> typeof(x)
Float32
```

Hexadecimal floating-point literals are also valid, but only as `Float64` values, with p preceding the base-2 exponent:

```
julia> 0x1p0
1.0

julia> 0x1.8p3
12.0

julia> x = 0x.4p-1
0.125

julia> typeof(x)
Float64
```

Half-precision floating-point numbers are also supported (`Float16`) on all platforms, with native instructions used on hardware which supports this number format. Otherwise, operations are implemented in software, and use `Float32` for intermediate calculations. As an internal implementation detail, this is achieved under the hood by using LLVM's `half` type, which behaves similarly to what the GCC `-fexcess-precision=16` flag does for C/C++ code.

```
julia> sizeof(Float16(4.))
2

julia> 2*Float16(4.)
Float16(8.0)
```

The underscore _ can be used as digit separator:

```
julia> 10_000, 0.000_000_005, 0xdead_beef, 0b1011_0010
(10000, 5.0e-9, 0xdeadbeef, 0xb2)
```

Floating-point zero

Floating-point numbers have two zeros, positive zero and negative zero. They are equal to each other but have different binary representations, as can be seen using the `bitstring` function:

```
julia> 0.0 == -0.0
true

julia> bitstring(0.0)
"0000000000000000000000000000000000000000000000000000000000000000"

julia> bitstring(-0.0)
"1000000000000000000000000000000000000000000000000000000000000000"
```

Float16	Float32	Float64	Name	Description
Inf16	Inf32	Inf	positive infinity	a value greater than all finite floating-point values
-Inf16	-Inf32	-Inf	negative infinity	a value less than all finite floating-point values
NaN16	NaN32	NaN	not a number	a value not == to any floating-point value (including itself)

Special floating-point values

There are three specified standard floating-point values that do not correspond to any point on the real number line:

For further discussion of how these non-finite floating-point values are ordered with respect to each other and other floats, see Numeric Comparisons. By the IEEE 754 standard, these floating-point values are the results of certain arithmetic operations:

```
julia> 1/Inf
0.0

julia> 1/0
Inf

julia> -5/0
-Inf

julia> 0.000001/0
Inf

julia> 0/0
NaN

julia> 500 + Inf
Inf

julia> 500 - Inf
-Inf

julia> Inf + Inf
Inf

julia> Inf - Inf
NaN

julia> Inf * Inf
Inf

julia> Inf / Inf
NaN

julia> 0 * Inf
```

```
NaN

julia> NaN == NaN
false

julia> NaN != NaN
true

julia> NaN < NaN
false

julia> NaN > NaN
false
```

The `typemin` and `typemax` functions also apply to floating-point types:

```
julia> (typemin(Float16),typemax(Float16))
(-Inf16, Inf16)

julia> (typemin(Float32),typemax(Float32))
(-Inf32, Inf32)

julia> (typemin(Float64),typemax(Float64))
(-Inf, Inf)
```

Machine epsilon

Most real numbers cannot be represented exactly with floating-point numbers, and so for many purposes it is important to know the distance between two adjacent representable floating-point numbers, which is often known as machine epsilon.

Julia provides `eps`, which gives the distance between `1.0` and the next larger representable floating-point value:

```
julia> eps(Float32)
1.1920929f-7

julia> eps(Float64)
2.220446049250313e-16

julia> eps() # same as eps(Float64)
2.220446049250313e-16
```

These values are `2.0^-23` and `2.0^-52` as `Float32` and `Float64` values, respectively. The `eps` function can also take a floating-point value as an argument, and gives the absolute difference between that value and the next representable floating point value. That is, `eps(x)` yields a value of the same type as `x` such that `x + eps(x)` is the next representable floating-point value larger than `x`:

```
julia> eps(1.0)
2.220446049250313e-16
```

```
julia> eps(1000.)
1.1368683772161603e-13

julia> eps(1e-27)
1.793662034335766e-43

julia> eps(0.0)
5.0e-324
```

The distance between two adjacent representable floating-point numbers is not constant, but is smaller for smaller values and larger for larger values. In other words, the representable floating-point numbers are densest in the real number line near zero, and grow sparser exponentially as one moves farther away from zero. By definition, `eps(1.0)` is the same as `eps(Float64)` since `1.0` is a 64-bit floating-point value.

Julia also provides the `nextfloat` and `prevfloat` functions which return the next largest or smallest representable floating-point number to the argument respectively:

```
julia> x = 1.25f0
1.25f0

julia> nextfloat(x)
1.2500001f0

julia> prevfloat(x)
1.2499999f0

julia> bitstring(prevfloat(x))
"00111111100111111111111111111111"

julia> bitstring(x)
"00111111101000000000000000000000"

julia> bitstring(nextfloat(x))
"00111111101000000000000000000001"
```

This example highlights the general principle that the adjacent representable floating-point numbers also have adjacent binary integer representations.

Rounding modes

If a number doesn't have an exact floating-point representation, it must be rounded to an appropriate representable value. However, the manner in which this rounding is done can be changed if required according to the rounding modes presented in the IEEE 754 standard.

The default mode used is always `RoundNearest`, which rounds to the nearest representable value, with ties rounded towards the nearest value with an even least significant bit.

Background and References

Floating-point arithmetic entails many subtleties which can be surprising to users who are unfamiliar with the low-level implementation details. However, these subtleties are described in detail in most books on scientific computation, and also in the following references:

- The definitive guide to floating point arithmetic is the IEEE 754-2008 Standard; however, it is not available for free online.
- For a brief but lucid presentation of how floating-point numbers are represented, see John D. Cook's article on the subject as well as his introduction to some of the issues arising from how this representation differs in behavior from the idealized abstraction of real numbers.
- Also recommended is Bruce Dawson's series of blog posts on floating-point numbers.
- For an excellent, in-depth discussion of floating-point numbers and issues of numerical accuracy encountered when computing with them, see David Goldberg's paper What Every Computer Scientist Should Know About Floating-Point Arithmetic.
- For even more extensive documentation of the history of, rationale for, and issues with floating-point numbers, as well as discussion of many other topics in numerical computing, see the collected writings of William Kahan, commonly known as the "Father of Floating-Point". Of particular interest may be An Interview with the Old Man of Floating-Point.

5.3 Arbitrary Precision Arithmetic

To allow computations with arbitrary-precision integers and floating point numbers, Julia wraps the GNU Multiple Precision Arithmetic Library (GMP) and the GNU MPFR Library, respectively. The `BigInt` and `BigFloat` types are available in Julia for arbitrary precision integer and floating point numbers respectively.

Constructors exist to create these types from primitive numerical types, and the string literal `@big_str` or `parse` can be used to construct them from `AbstractStrings`. `BigInts` can also be input as integer literals when they are too big for other built-in integer types. Note that as there is no unsigned arbitrary-precision integer type in `Base` (`BigInt` is sufficient in most cases), hexadecimal, octal and binary literals can be used (in addition to decimal literals).

Once created, they participate in arithmetic with all other numeric types thanks to Julia's type promotion and conversion mechanism:

```
julia> BigInt(typemax(Int64)) + 1
9223372036854775808

julia> big"123456789012345678901234567890" + 1
123456789012345678901234567891

julia> parse(BigInt, "123456789012345678901234567890") + 1
123456789012345678901234567891

julia> string(big"2"^200, base=16)
"100000000000000000000000000000000000000000000000000"

julia> 0x100000000000000000000000000000000-1 == typemax(UInt128)
true

julia> 0x00000000000000000000000000000000
0

julia> typeof(ans)
BigInt
```

```
julia> big"1.23456789012345678901"
1.2345678901234567890100000000000000000000000000000000000000000000000000000000000000000004

julia> parse(BigFloat, "1.23456789012345678901")
1.2345678901234567890100000000000000000000000000000000000000000000000000000000000000000004

julia> BigFloat(2.0^66) / 3
2.4595658764946068821333333333333333333333333333333333333333333333333333333333333333344e19

julia> factorial(BigInt(40))
815915283247897734345611269596115894272000000000
```

However, type promotion between the primitive types above and `BigInt`/`BigFloat` is not automatic and must be explicitly stated.

```
julia> x = typemin(Int64)
-9223372036854775808

julia> x = x - 1
9223372036854775807

julia> typeof(x)
Int64

julia> y = BigInt(typemin(Int64))
-9223372036854775808

julia> y = y - 1
-9223372036854775809

julia> typeof(y)
BigInt
```

The default precision (in number of bits of the significand) and rounding mode of `BigFloat` operations can be changed globally by calling `setprecision` and `setrounding`, and all further calculations will take these changes in account. Alternatively, the precision or the rounding can be changed only within the execution of a particular block of code by using the same functions with a `do` block:

```
julia> setrounding(BigFloat, RoundUp) do
           BigFloat(1) + parse(BigFloat, "0.1")
       end
1.1000000000000000000000000000000000000000000000000000000000000000000000000000000000000003

julia> setrounding(BigFloat, RoundDown) do
           BigFloat(1) + parse(BigFloat, "0.1")
       end
1.0999999999999999999999999999999999999999999999999999999999999999999999999999999999999986

julia> setprecision(40) do
           BigFloat(1) + parse(BigFloat, "0.1")
```

```
    end
1.1000000000004
```

Warning

The relation between `setprecision` or `setrounding` and `@big_str`, the macro used for `big` string literals (such as `big"0.3"`), might not be intuitive, as a consequence of the fact that `@big_str` is a macro. See the `@big_str` documentation for details.

5.4 Numeric Literal Coefficients

To make common numeric formulae and expressions clearer, Julia allows variables to be immediately preceded by a numeric literal, implying multiplication. This makes writing polynomial expressions much cleaner:

```
julia> x = 3
3

julia> 2x^2 - 3x + 1
10

julia> 1.5x^2 - .5x + 1
13.0
```

It also makes writing exponential functions more elegant:

```
julia> 2^2x
64
```

The precedence of numeric literal coefficients is slightly lower than that of unary operators such as negation. So `-2x` is parsed as `(-2) * x` and `√2x` is parsed as `(√2) * x`. However, numeric literal coefficients parse similarly to unary operators when combined with exponentiation. For example `2^3x` is parsed as `2^(3x)`, and `2x^3` is parsed as `2*(x^3)`.

Numeric literals also work as coefficients to parenthesized expressions:

```
julia> 2(x-1)^2 - 3(x-1) + 1
3
```

Note

The precedence of numeric literal coefficients used for implicit multiplication is higher than other binary operators such as multiplication (`*`), and division (`/`, `\`, and `//`). This means, for example, that `1 / 2im` equals `-0.5im` and `6 // 2(2 + 1)` equals `1 // 1`.

Additionally, parenthesized expressions can be used as coefficients to variables, implying multiplication of the expression by the variable:

```
julia> (x-1)x
6
```

Neither juxtaposition of two parenthesized expressions, nor placing a variable before a parenthesized expression, however, can be used to imply multiplication:

```
julia> (x-1)(x+1)
ERROR: MethodError: objects of type Int64 are not callable

julia> x(x+1)
ERROR: MethodError: objects of type Int64 are not callable
```

Both expressions are interpreted as function application: any expression that is not a numeric literal, when immediately followed by a parenthetical, is interpreted as a function applied to the values in parentheses (see Functions for more about functions). Thus, in both of these cases, an error occurs since the left-hand value is not a function.

The above syntactic enhancements significantly reduce the visual noise incurred when writing common mathematical formulae. Note that no whitespace may come between a numeric literal coefficient and the identifier or parenthesized expression which it multiplies.

Syntax Conflicts

Juxtaposed literal coefficient syntax may conflict with some numeric literal syntaxes: hexadecimal, octal and binary integer literals and engineering notation for floating-point literals. Here are some situations where syntactic conflicts arise:

- The hexadecimal integer literal expression `0xff` could be interpreted as the numeric literal `0` multiplied by the variable `xff`. Similar ambiguities arise with octal and binary literals like `0o777` or `0b01001010`.
- The floating-point literal expression `1e10` could be interpreted as the numeric literal `1` multiplied by the variable `e10`, and similarly with the equivalent E form.
- The 32-bit floating-point literal expression `1.5f22` could be interpreted as the numeric literal `1.5` multiplied by the variable `f22`.

In all cases the ambiguity is resolved in favor of interpretation as numeric literals:

- Expressions starting with `0x`/`0o`/`0b` are always hexadecimal/octal/binary literals.
- Expressions starting with a numeric literal followed by `e` or `E` are always floating-point literals.
- Expressions starting with a numeric literal followed by `f` are always 32-bit floating-point literals.

Unlike E, which is equivalent to e in numeric literals for historical reasons, F is just another letter and does not behave like `f` in numeric literals. Hence, expressions starting with a numeric literal followed by F are interpreted as the numerical literal multiplied by a variable, which means that, for example, `1.5F22` is equal to `1.5 * F22`.

5.5 Literal zero and one

Julia provides functions which return literal 0 and 1 corresponding to a specified type or the type of a given variable.

Function	Description
`zero(x)`	Literal zero of type x or type of variable x
`one(x)`	Literal one of type x or type of variable x

These functions are useful in Numeric Comparisons to avoid overhead from unnecessary type conversion.

Examples:

```
julia> zero(Float32)
0.0f0

julia> zero(1.0)
0.0

julia> one(Int32)
1

julia> one(BigFloat)
1.0
```

Chapter 6

Mathematical Operations and Elementary Functions

Julia provides a complete collection of basic arithmetic and bitwise operators across all of its numeric primitive types, as well as providing portable, efficient implementations of a comprehensive collection of standard mathematical functions.

6.1 Arithmetic Operators

The following arithmetic operators are supported on all primitive numeric types:

Expression	Name	Description
`+x`	unary plus	the identity operation
`-x`	unary minus	maps values to their additive inverses
`x + y`	binary plus	performs addition
`x - y`	binary minus	performs subtraction
`x * y`	times	performs multiplication
`x / y`	divide	performs division
`x ÷ y`	integer divide	x / y, truncated to an integer
`x \ y`	inverse divide	equivalent to `y / x`
`x ^ y`	power	raises x to the yth power
`x % y`	remainder	equivalent to `rem(x, y)`

A numeric literal placed directly before an identifier or parentheses, e.g. `2x` or `2(x + y)`, is treated as a multiplication, except with higher precedence than other binary operations. See Numeric Literal Coefficients for details.

Julia's promotion system makes arithmetic operations on mixtures of argument types "just work" naturally and automatically. See Conversion and Promotion for details of the promotion system.

The ÷ sign can be conveniently typed by writing `\div<tab>` to the REPL or Julia IDE. See the manual section on Unicode input for more information.

Here are some simple examples using arithmetic operators:

```
julia> 1 + 2 + 3
```

```
6

julia> 1 - 2
-1

julia> 3*2/12
0.5
```

(By convention, we tend to space operators more tightly if they get applied before other nearby operators. For instance, we would generally write `-x + 2` to reflect that first `x` gets negated, and then 2 is added to that result.)

6.2 Boolean Operators

The following Boolean operators are supported on `Bool` types:

Expression	Name
`!x`	negation
`x && y`	short-circuiting and
`x || y`	short-circuiting or

Negation changes `true` to `false` and vice versa. The short-circuiting operations are explained on the linked page.

6.3 Arithmetic operations with `Bool` values

Note that `Bool` is an integer type, such that `false` is numerically equal to `0` and `true` is numerically equal to `1`. All the usual promotion rules and numeric operators are also defined on it, with a special behavior of arithmetic (non-Boolean) operations when all the arguments are `Bool`: in those cases, the arguments are promoted to `Int` instead of keeping their type. Compare e.g. the following equivalent operations with `Bool` and with a different numeric type (`UInt8`):

```
julia> true - true
0

julia> 0x01 - 0x01
0x00
```

Also, when used in multiplication, `false` acts as a *strong zero*:

```
julia> NaN * false
0.0

julia> false * Inf
0.0
```

This is useful for preventing the propagation of `NaN` values in quantities that are known to be zero. See Knuth (1992) for motivation.

6.4 Bitwise Operators

The following bitwise operators are supported on all primitive integer types:

Expression	Name
~x	bitwise not
x & y	bitwise and
x | y	bitwise or
x ⊻ y	bitwise xor (exclusive or)
x ⊼ y	bitwise nand (not and)
x ⊽ y	bitwise nor (not or)
x >>> y	logical shift right
x >> y	arithmetic shift right
x << y	logical/arithmetic shift left

Here are some examples with bitwise operators:

```
julia> ~123
-124

julia> 123 & 234
106

julia> 123 | 234
251

julia> 123 ⊻ 234
145

julia> xor(123, 234)
145

julia> nand(123, 123)
-124

julia> 123 ⊼ 123
-124

julia> nor(123, 124)
-128

julia> 123 ⊽ 124
-128

julia> ~UInt32(123)
0xffffff84

julia> ~UInt8(123)
0x84
```

6.5 Updating operators

Every binary arithmetic and bitwise operator also has an updating version that assigns the result of the operation back into its left operand. The updating version of the binary operator is formed by placing a = immediately after the operator. For example, writing `x += 3` is equivalent to writing `x = x + 3`:

```
julia> x = 1
1

julia> x += 3
4

julia> x
4
```

The updating versions of all the binary arithmetic and bitwise operators are:

```
+=  -=  *=  /=  \=  ÷=  %=  ^=  &=  |=  ⊻=  >>>=  >>=  <<=
```

Note

An updating operator rebinds the variable on the left-hand side. As a result, the type of the variable may change.

```
julia> x = 0x01; typeof(x)
UInt8

julia> x *= 2 # Same as x = x * 2
2

julia> typeof(x)
Int64
```

6.6 Vectorized "dot" operators

For *every* binary operation like ^, there is a corresponding "dot" operation `.^` that is *automatically* defined to perform ^ element-by-element on arrays. For example, `[1, 2, 3] ^ 3` is not defined, since there is no standard mathematical meaning to "cubing" a (non-square) array, but `[1, 2, 3] .^ 3` is defined as computing the elementwise (or "vectorized") result `[1^3, 2^3, 3^3]`. Similarly for unary operators like ! or √, there is a corresponding `.√` that applies the operator elementwise.

```
julia> [1, 2, 3] .^ 3
3-element Vector{Int64}:
  1
  8
 27
```

More specifically, `a .^ b` is parsed as the "dot" call `(^).(a,b)`, which performs a broadcast operation: it can combine arrays and scalars, arrays of the same size (performing the operation elementwise), and even arrays of different shapes (e.g. combining row and column vectors to produce a matrix). Moreover, like all vectorized "dot calls," these "dot operators" are *fusing*. For example, if you compute `2 .* A.^2 .+ sin.(A)` (or equivalently `@. 2A^2 + sin(A)`, using the `@.` macro) for an array `A`, it performs a *single* loop over `A`, computing `2a^2 + sin(a)` for each element `a` of `A`. In particular, nested dot calls like `f.(g.(x))` are fused, and "adjacent" binary operators like `x .+ 3 .* x.^2` are equivalent to nested dot calls `(+).(x, (*).(3, (^).(x, 2)))`.

Furthermore, "dotted" updating operators like `a .+= b` (or `@. a += b`) are parsed as `a .= a .+ b`, where `.=` is a fused *in-place* assignment operation (see the dot syntax documentation).

Note the dot syntax is also applicable to user-defined operators. For example, if you define `⊗(A, B) = kron(A, B)` to give a convenient infix syntax `A ⊗ B` for Kronecker products (`kron`), then `[A, B] .⊗ [C, D]` will compute `[A⊗C, B⊗D]` with no additional coding.

Combining dot operators with numeric literals can be ambiguous. For example, it is not clear whether `1.+x` means `1. + x` or `1 .+ x`. Therefore this syntax is disallowed, and spaces must be used around the operator in such cases.

6.7 Numeric Comparisons

Standard comparison operations are defined for all the primitive numeric types:

Operator	Name
`==`	equality
`!=`, `≠`	inequality
`<`	less than
`<=`, `≤`	less than or equal to
`>`	greater than
`>=`, `≥`	greater than or equal to

Here are some simple examples:

```
julia> 1 == 1
true

julia> 1 == 2
false

julia> 1 != 2
true

julia> 1 == 1.0
true

julia> 1 < 2
true

julia> 1.0 > 3
false
```

```
julia> 1 >= 1.0
true

julia> -1 <= 1
true

julia> -1 <= -1
true

julia> -1 <= -2
false

julia> 3 < -0.5
false
```

Integers are compared in the standard manner – by comparison of bits. Floating-point numbers are compared according to the IEEE 754 standard:

- Finite numbers are ordered in the usual manner.
- Positive zero is equal but not greater than negative zero.
- `Inf` is equal to itself and greater than everything else except `NaN`.
- `-Inf` is equal to itself and less than everything else except `NaN`.
- `NaN` is not equal to, not less than, and not greater than anything, including itself.

The last point is potentially surprising and thus worth noting:

```
julia> NaN == NaN
false

julia> NaN != NaN
true

julia> NaN < NaN
false

julia> NaN > NaN
false
```

and can cause headaches when working with arrays:

```
julia> [1 NaN] == [1 NaN]
false
```

Julia provides additional functions to test numbers for special values, which can be useful in situations like hash key comparisons:

`isequal` considers `NaN`s equal to each other:

Function	Tests if
`isequal(x, y)`	x and y are identical
`isfinite(x)`	x is a finite number
`isinf(x)`	x is infinite
`isnan(x)`	x is not a number

```
julia> isequal(NaN, NaN)
true

julia> isequal([1 NaN], [1 NaN])
true

julia> isequal(NaN, NaN32)
true
```

`isequal` can also be used to distinguish signed zeros:

```
julia> -0.0 == 0.0
true

julia> isequal(-0.0, 0.0)
false
```

Mixed-type comparisons between signed integers, unsigned integers, and floats can be tricky. A great deal of care has been taken to ensure that Julia does them correctly.

For other types, `isequal` defaults to calling `==`, so if you want to define equality for your own types then you only need to add a `==` method. If you define your own equality function, you should probably define a corresponding `hash` method to ensure that `isequal(x,y)` implies `hash(x) == hash(y)`.

Chaining comparisons

Unlike most languages, with the notable exception of Python, comparisons can be arbitrarily chained:

```
julia> 1 < 2 <= 2 < 3 == 3 > 2 >= 1 == 1 < 3 != 5
true
```

Chaining comparisons is often quite convenient in numerical code. Chained comparisons use the && operator for scalar comparisons, and the & operator for elementwise comparisons, which allows them to work on arrays. For example, `0 .< A .< 1` gives a boolean array whose entries are true where the corresponding elements of `A` are between 0 and 1.

Note the evaluation behavior of chained comparisons:

```
julia> v(x) = (println(x); x)
v (generic function with 1 method)

julia> v(1) < v(2) <= v(3)
2
```

```
1
3
true

julia> v(1) > v(2) <= v(3)
2
1
false
```

The middle expression is only evaluated once, rather than twice as it would be if the expression were written as `v(1) < v(2) && v(2) <= v(3)`. However, the order of evaluations in a chained comparison is undefined. It is strongly recommended not to use expressions with side effects (such as printing) in chained comparisons. If side effects are required, the short-circuit && operator should be used explicitly (see Short-Circuit Evaluation).

Elementary Functions

Julia provides a comprehensive collection of mathematical functions and operators. These mathematical operations are defined over as broad a class of numerical values as permit sensible definitions, including integers, floating-point numbers, rationals, and complex numbers, wherever such definitions make sense.

Moreover, these functions (like any Julia function) can be applied in "vectorized" fashion to arrays and other collections with the dot syntax `f.(A)`, e.g. `sin.(A)` will compute the sine of each element of an array `A`.

6.8 Operator Precedence and Associativity

Julia applies the following order and associativity of operations, from highest precedence to lowest:

Category	Operators	Associativity
Syntax	`.` followed by `::`	Left
Exponentiation	`^`	Right
Unary	`+ - ! ~ ¬ √ ∛ ∜ * ± ∓ <: >:`	Right[1]
Bitshifts	`<< >> >>>`	Left
Fractions	`//`	Left
Multiplication	`* / % & \ ÷`	Left[2]
Addition	`+ - | ⊻`	Left[2]
Syntax	`: ..`	Left
Syntax	`|>`	Left
Syntax	`<|`	Right
Comparisons	`> < >= <= == === != !== <:`	Non-associative
Control flow	&& followed by || followed by ?	Right
Pair	`=>`	Right
Assignments	`= += -= *= /= //= \= ^= ÷= %= |= &= ⊻= <<= >>= >>>=`	Right

[1] The unary operators + and - require explicit parentheses around their argument to disambiguate them from the operator ++, etc. Other compositions of unary operators are parsed with right-associativity, e. g., √√-a as √(√(-a)).

[2] The operators +, ++ and * are non-associative. `a + b + c` is parsed as `+(a, b, c)` not `+(+(a, b), c)`. However, the fallback methods for `+(a, b, c, d...)` and `*(a, b, c, d...)` both default to left-associative evaluation.

For a complete list of *every* Julia operator's precedence, see the top of this file: `src/julia-parser.scm`. Note that some of the operators there are not defined in the Base module but may be given definitions by standard libraries, packages or user code.

You can also find the numerical precedence for any given operator via the built-in function `Base.operator_precedence`, where higher numbers take precedence:

```
julia> Base.operator_precedence(:+), Base.operator_precedence(:*), Base.operator_precedence(:.)
(11, 12, 17)

julia> Base.operator_precedence(:sin), Base.operator_precedence(:+=),
↪  Base.operator_precedence(:(=))  # (Note the necessary parens on `:(=)`)
(0, 1, 1)
```

A symbol representing the operator associativity can also be found by calling the built-in function `Base.operator_associativity`:

```
julia> Base.operator_associativity(:-), Base.operator_associativity(:+),
↪  Base.operator_associativity(:^)
(:left, :none, :right)

julia> Base.operator_associativity(:⊗), Base.operator_associativity(:sin),
↪  Base.operator_associativity(:→)
(:left, :none, :right)
```

Note that symbols such as `:sin` return precedence `0`. This value represents invalid operators and not operators of lowest precedence. Similarly, such operators are assigned associativity `:none`.

Numeric literal coefficients, e.g. `2x`, are treated as multiplications with higher precedence than any other binary operation, with the exception of `^` where they have higher precedence only as the exponent.

```
julia> x = 3; 2x^2
18

julia> x = 3; 2^2x
64
```

Juxtaposition parses like a unary operator, which has the same natural asymmetry around exponents: `-x^y` and `2x^y` parse as `-(x^y)` and `2(x^y)` whereas `x^-y` and `x^2y` parse as `x^(-y)` and `x^(2y)`.

6.9 Numerical Conversions

Julia supports three forms of numerical conversion, which differ in their handling of inexact conversions.

- The notation `T(x)` or `convert(T, x)` converts `x` to a value of type `T`.
  - If `T` is a floating-point type, the result is the nearest representable value, which could be positive or negative infinity.
  - If `T` is an integer type, an `InexactError` is raised if `x` is not representable by `T`.

- `x % T` converts an integer `x` to a value of integer type `T` congruent to `x` modulo `2^n`, where `n` is the number of bits in `T`. In other words, the binary representation is truncated to fit.
- The Rounding functions take a type `T` as an optional argument. For example, `round(Int,x)` is a shorthand for `Int(round(x))`.

The following examples show the different forms.

```
julia> Int8(127)
127

julia> Int8(128)
ERROR: InexactError: trunc(Int8, 128)
Stacktrace:
[...]

julia> Int8(127.0)
127

julia> Int8(3.14)
ERROR: InexactError: Int8(3.14)
Stacktrace:
[...]

julia> Int8(128.0)
ERROR: InexactError: Int8(128.0)
Stacktrace:
[...]

julia> 127 % Int8
127

julia> 128 % Int8
-128

julia> round(Int8,127.4)
127

julia> round(Int8,127.6)
ERROR: InexactError: Int8(128.0)
Stacktrace:
[...]
```

See Conversion and Promotion for how to define your own conversions and promotions.

Function	Description	Return type
`round(x)`	round x to the nearest integer	`typeof(x)`
`round(T, x)`	round x to the nearest integer	T
`floor(x)`	round x towards `-Inf`	`typeof(x)`
`floor(T, x)`	round x towards `-Inf`	T
`ceil(x)`	round x towards `+Inf`	`typeof(x)`
`ceil(T, x)`	round x towards `+Inf`	T
`trunc(x)`	round x towards zero	`typeof(x)`
`trunc(T, x)`	round x towards zero	T

Function	Description
`div(x, y)`, x÷y	truncated division; quotient rounded towards zero
`fld(x, y)`	floored division; quotient rounded towards `-Inf`
`cld(x, y)`	ceiling division; quotient rounded towards `+Inf`
`rem(x, y)`, x%y	remainder; satisfies `x == div(x, y)*y + rem(x, y)`; sign matches x
`mod(x, y)`	modulus; satisfies `x == fld(x, y)*y + mod(x, y)`; sign matches y
`mod1(x, y)`	`mod` with offset 1; returns `r∈(0, y]` for `y>0` or `r∈[y, 0)` for `y<0`, where `mod(r, y) == mod(x, y)`
`mod2pi(x)`	modulus with respect to 2pi; `0 <= mod2pi(x) < 2pi`
`divrem(x, y)`	returns `(div(x, y),rem(x, y))`
`fldmod(x, y)`	returns `(fld(x, y), mod(x, y))`
`gcd(x, y...)`	greatest positive common divisor of x, y,...
`lcm(x, y...)`	least positive common multiple of x, y,...

Function	Description
`abs(x)`	a positive value with the magnitude of x
`abs2(x)`	the squared magnitude of x
`sign(x)`	indicates the sign of x, returning -1, 0, or +1
`signbit(x)`	indicates whether the sign bit is on (true) or off (false)
`copysign(x, y)`	a value with the magnitude of x and the sign of y
`flipsign(x, y)`	a value with the magnitude of x and the sign of x*y

Rounding functions

Division functions

Sign and absolute value functions

Powers, logs and roots

For an overview of why functions like `hypot`, `expm1`, and `log1p` are necessary and useful, see John D. Cook's excellent pair of blog posts on the subject: expm1, log1p, erfc, and hypot.

Function	Description
`sqrt(x)`, √x	square root of x
`cbrt(x)`, ∛x	cube root of x
`fourthroot(x)`, ∜x	fourth root of x
`hypot(x, y)`	hypotenuse of right-angled triangle with other sides of length x and y
`exp(x)`	natural exponential function at x
`expm1(x)`	accurate `exp(x) - 1` for x near zero
`ldexp(x, n)`	`x * 2^n` computed efficiently for integer values of n
`log(x)`	natural logarithm of x
`log(b, x)`	base b logarithm of x
`log2(x)`	base 2 logarithm of x
`log10(x)`	base 10 logarithm of x
`log1p(x)`	accurate `log(1 + x)` for x near zero
`exponent(x)`	binary exponent of x
`significand(x)`	binary significand (a.k.a. mantissa) of a floating-point number x

Trigonometric and hyperbolic functions

All the standard trigonometric and hyperbolic functions are also defined:

```
sin    cos    tan    cot    sec    csc
sinh   cosh   tanh   coth   sech   csch
asin   acos   atan   acot   asec   acsc
asinh  acosh  atanh  acoth  asech  acsch
sinc   cosc
```

These are all single-argument functions, with `atan` also accepting two arguments corresponding to a traditional `atan2` function.

Additionally, `sinpi(x)` and `cospi(x)` are provided for more accurate computations of `sin(pi * x)` and `cos(pi * x)` respectively.

In order to compute trigonometric functions with degrees instead of radians, suffix the function with `d`. For example, `sind(x)` computes the sine of x where x is specified in degrees. The complete list of trigonometric functions with degree variants is:

```
sind   cosd   tand   cotd   secd   cscd
asind  acosd  atand  acotd  asecd  acscd
```

Special functions

Many other special mathematical functions are provided by the package SpecialFunctions.jl.

Chapter 7

Complex and Rational Numbers

Julia includes predefined types for both complex and rational numbers, and supports all the standard Mathematical Operations and Elementary Functions on them. Conversion and Promotion are defined so that operations on any combination of predefined numeric types, whether primitive or composite, behave as expected.

7.1 Complex Numbers

The global constant `im` is bound to the complex number *i*, representing the principal square root of -1. (Using mathematicians' `i` or engineers' `j` for this global constant was rejected since they are such popular index variable names.) Since Julia allows numeric literals to be juxtaposed with identifiers as coefficients, this binding suffices to provide convenient syntax for complex numbers, similar to the traditional mathematical notation:

```
julia> 1+2im
1 + 2im
```

You can perform all the standard arithmetic operations with complex numbers:

```
julia> (1 + 2im)*(2 - 3im)
8 + 1im

julia> (1 + 2im)/(1 - 2im)
-0.6 + 0.8im

julia> (1 + 2im) + (1 - 2im)
2 + 0im

julia> (-3 + 2im) - (5 - 1im)
-8 + 3im

julia> (-1 + 2im)^2
-3 - 4im

julia> (-1 + 2im)^2.5
2.729624464784009 - 6.9606644595719im
```

```
julia> (-1 + 2im)^(1 + 1im)
-0.27910381075826657 + 0.08708053414102428im

julia> 3(2 - 5im)
6 - 15im

julia> 3(2 - 5im)^2
-63 - 60im

julia> 3(2 - 5im)^-1.0
0.20689655172413793 + 0.5172413793103449im
```

The promotion mechanism ensures that combinations of operands of different types just work:

```
julia> 2(1 - 1im)
2 - 2im

julia> (2 + 3im) - 1
1 + 3im

julia> (1 + 2im) + 0.5
1.5 + 2.0im

julia> (2 + 3im) - 0.5im
2.0 + 2.5im

julia> 0.75(1 + 2im)
0.75 + 1.5im

julia> (2 + 3im) / 2
1.0 + 1.5im

julia> (1 - 3im) / (2 + 2im)
-0.5 - 1.0im

julia> 2im^2
-2 + 0im

julia> 1 + 3/4im
1.0 - 0.75im
```

Note that `3/4im == 3/(4*im) == -(3/4*im)`, since a literal coefficient binds more tightly than division.

Standard functions to manipulate complex values are provided:

```
julia> z = 1 + 2im
1 + 2im

julia> real(1 + 2im) # real part of z
1
```

```
julia> imag(1 + 2im) # imaginary part of z
2

julia> conj(1 + 2im) # complex conjugate of z
1 - 2im

julia> abs(1 + 2im) # absolute value of z
2.23606797749979

julia> abs2(1 + 2im) # squared absolute value
5

julia> angle(1 + 2im) # phase angle in radians
1.1071487177940904
```

As usual, the absolute value (`abs`) of a complex number is its distance from zero. `abs2` gives the square of the absolute value, and is of particular use for complex numbers since it avoids taking a square root. `angle` returns the phase angle in radians (also known as the *argument* or *arg* function). The full gamut of other Elementary Functions is also defined for complex numbers:

```
julia> sqrt(1im)
0.7071067811865476 + 0.7071067811865475im

julia> sqrt(1 + 2im)
1.272019649514069 + 0.7861513777574233im

julia> cos(1 + 2im)
2.0327230070196656 - 3.0518977991517997im

julia> exp(1 + 2im)
-1.1312043837568135 + 2.4717266720048188im

julia> sinh(1 + 2im)
-0.4890562590412937 + 1.4031192506220405im
```

Note that mathematical functions typically return real values when applied to real numbers and complex values when applied to complex numbers. For example, `sqrt` behaves differently when applied to `-1` versus `-1 + 0im` even though `-1 == -1 + 0im`:

```
julia> sqrt(-1)
ERROR: DomainError with -1.0:
sqrt was called with a negative real argument but will only return a complex result if called
↪  with a complex argument. Try sqrt(Complex(x)).
Stacktrace:
[...]

julia> sqrt(-1 + 0im)
0.0 + 1.0im
```

The literal numeric coefficient notation does not work when constructing a complex number from variables. Instead, the multiplication must be explicitly written out:

```
julia> a = 1; b = 2; a + b*im
1 + 2im
```

However, this is *not* recommended. Instead, use the more efficient `complex` function to construct a complex value directly from its real and imaginary parts:

```
julia> a = 1; b = 2; complex(a, b)
1 + 2im
```

This construction avoids the multiplication and addition operations.

`Inf` and `NaN` propagate through complex numbers in the real and imaginary parts of a complex number as described in the Special floating-point values section:

```
julia> 1 + Inf*im
1.0 + Inf*im

julia> 1 + NaN*im
1.0 + NaN*im
```

7.2 Rational Numbers

Julia has a rational number type to represent exact ratios of integers. Rationals are constructed using the `//` operator:

```
julia> 2//3
2//3
```

If the numerator and denominator of a rational have common factors, they are reduced to lowest terms such that the denominator is non-negative:

```
julia> 6//9
2//3

julia> -4//8
-1//2

julia> 5//-15
-1//3

julia> -4//-12
1//3
```

This normalized form for a ratio of integers is unique, so equality of rational values can be tested by checking for equality of the numerator and denominator. The standardized numerator and denominator of a rational value can be extracted using the `numerator` and `denominator` functions:

```
julia> numerator(2//3)
2

julia> denominator(2//3)
3
```

Direct comparison of the numerator and denominator is generally not necessary, since the standard arithmetic and comparison operations are defined for rational values:

```
julia> 2//3 == 6//9
true

julia> 2//3 == 9//27
false

julia> 3//7 < 1//2
true

julia> 3//4 > 2//3
true

julia> 2//4 + 1//6
2//3

julia> 5//12 - 1//4
1//6

julia> 5//8 * 3//12
5//32

julia> 6//5 / 10//7
21//25
```

Rationals can easily be converted to floating-point numbers:

```
julia> float(3//4)
0.75
```

Conversion from rational to floating-point respects the following identity for any integral values of a and b, except when `a==0 && b <= 0`:

```
julia> a = 1; b = 2;

julia> isequal(float(a//b), a/b)
true

julia> a, b = 0, 0
(0, 0)

julia> float(a//b)
ERROR: ArgumentError: invalid rational: zero(Int64)//zero(Int64)
```

```
Stacktrace:
[...]

julia> a/b
NaN

julia> a, b = 0, -1
(0, -1)

julia> float(a//b), a/b
(0.0, -0.0)
```

Constructing infinite rational values is acceptable:

```
julia> 5//0
1//0

julia> x = -3//0
-1//0

julia> typeof(x)
Rational{Int64}
```

Trying to construct a `NaN` rational value, however, is invalid:

```
julia> 0//0
ERROR: ArgumentError: invalid rational: zero(Int64)//zero(Int64)
Stacktrace:
[...]
```

As usual, the promotion system makes interactions with other numeric types effortless:

```
julia> 3//5 + 1
8//5

julia> 3//5 - 0.5
0.09999999999999998

julia> 2//7 * (1 + 2im)
2//7 + 4//7*im

julia> 2//7 * (1.5 + 2im)
0.42857142857142855 + 0.5714285714285714im

julia> 3//2 / (1 + 2im)
3//10 - 3//5*im

julia> 1//2 + 2im
1//2 + 2//1*im

julia> 1 + 2//3im
```

```
1//1 - 2//3*im

julia> 0.5 == 1//2
true

julia> 0.33 == 1//3
false

julia> 0.33 < 1//3
true

julia> 1//3 - 0.33
0.0033333333333332993
```

Chapter 8

Strings

Strings are finite sequences of characters. Of course, the real trouble comes when one asks what a character is. The characters that English speakers are familiar with are the letters A, B, C, etc., together with numerals and common punctuation symbols. These characters are standardized together with a mapping to integer values between 0 and 127 by the ASCII standard. There are, of course, many other characters used in non-English languages, including variants of the ASCII characters with accents and other modifications, related scripts such as Cyrillic and Greek, and scripts completely unrelated to ASCII and English, including Arabic, Chinese, Hebrew, Hindi, Japanese, and Korean. The Unicode standard tackles the complexities of what exactly a character is, and is generally accepted as the definitive standard addressing this problem. Depending on your needs, you can either ignore these complexities entirely and just pretend that only ASCII characters exist, or you can write code that can handle any of the characters or encodings that one may encounter when handling non-ASCII text. Julia makes dealing with plain ASCII text simple and efficient, and handling Unicode is as simple and efficient as possible. In particular, you can write C-style string code to process ASCII strings, and they will work as expected, both in terms of performance and semantics. If such code encounters non-ASCII text, it will gracefully fail with a clear error message, rather than silently introducing corrupt results. When this happens, modifying the code to handle non-ASCII data is straightforward.

There are a few noteworthy high-level features about Julia's strings:

- The built-in concrete type used for strings (and string literals) in Julia is `String`. This supports the full range of Unicode characters via the UTF-8 encoding. (A `transcode` function is provided to convert to/from other Unicode encodings.)
- All string types are subtypes of the abstract type `AbstractString`, and external packages define additional `AbstractString` subtypes (e.g. for other encodings). If you define a function expecting a string argument, you should declare the type as `AbstractString` in order to accept any string type.
- Like C and Java, but unlike most dynamic languages, Julia has a first-class type for representing a single character, called `AbstractChar`. The built-in `Char` subtype of `AbstractChar` is a 32-bit primitive type that can represent any Unicode character (and which is based on the UTF-8 encoding).
- As in Java, strings are immutable: the value of an `AbstractString` object cannot be changed. To construct a different string value, you construct a new string from parts of other strings.
- Conceptually, a string is a *partial function* from indices to characters: for some index values, no character value is returned, and instead an exception is thrown. This allows for efficient indexing

into strings by the byte index of an encoded representation rather than by a character index, which cannot be implemented both efficiently and simply for variable-width encodings of Unicode strings.

8.1 Characters

A `Char` value represents a single character: it is just a 32-bit primitive type with a special literal representation and appropriate arithmetic behaviors, and which can be converted to a numeric value representing a Unicode code point. (Julia packages may define other subtypes of `AbstractChar`, e.g. to optimize operations for other text encodings.) Here is how `Char` values are input and shown (note that character literals are delimited with single quotes, not double quotes):

```
julia> c = 'x'
'x': ASCII/Unicode U+0078 (category Ll: Letter, lowercase)

julia> typeof(c)
Char
```

You can easily convert a `Char` to its integer value, i.e. code point:

```
julia> c = Int('x')
120

julia> typeof(c)
Int64
```

On 32-bit architectures, `typeof(c)` will be `Int32`. You can convert an integer value back to a `Char` just as easily:

```
julia> Char(120)
'x': ASCII/Unicode U+0078 (category Ll: Letter, lowercase)
```

Not all integer values are valid Unicode code points, but for performance, the `Char` conversion does not check that every character value is valid. If you want to check that each converted value is a valid code point, use the `isvalid` function:

```
julia> Char(0x110000)
'\U110000': Unicode U+110000 (category In: Invalid, too high)

julia> isvalid(Char, 0x110000)
false
```

As of this writing, the valid Unicode code points are `U+0000` through `U+D7FF` and `U+E000` through `U+10FFFF`. These have not all been assigned intelligible meanings yet, nor are they necessarily interpretable by applications, but all of these values are considered to be valid Unicode characters.

You can input any Unicode character in single quotes using `\u` followed by up to four hexadecimal digits or `\U` followed by up to eight hexadecimal digits (the longest valid value only requires six):

```
julia> '\u0'
'\0': ASCII/Unicode U+0000 (category Cc: Other, control)

julia> '\u78'
'x': ASCII/Unicode U+0078 (category Ll: Letter, lowercase)

julia> '\u2200'
'∀': Unicode U+2200 (category Sm: Symbol, math)

julia> '\U10ffff'
'\U10ffff': Unicode U+10FFFF (category Cn: Other, not assigned)
```

Julia uses your system's locale and language settings to determine which characters can be printed as-is and which must be output using the generic, escaped `\u` or `\U` input forms. In addition to these Unicode escape forms, all of C's traditional escaped input forms can also be used:

```
julia> Int('\0')
0

julia> Int('\t')
9

julia> Int('\n')
10

julia> Int('\e')
27

julia> Int('\x7f')
127

julia> Int('\177')
127
```

You can do comparisons and a limited amount of arithmetic with `Char` values:

```
julia> 'A' < 'a'
true

julia> 'A' <= 'a' <= 'Z'
false

julia> 'A' <= 'X' <= 'Z'
true

julia> 'x' - 'a'
23

julia> 'A' + 1
'B': ASCII/Unicode U+0042 (category Lu: Letter, uppercase)
```

8.2 String Basics

String literals are delimited by double quotes or triple double quotes (not single quotes):

```
julia> str = "Hello, world.\n"
"Hello, world.\n"

julia> """Contains "quote" characters"""
"Contains \"quote\" characters"
```

Long lines in strings can be broken up by preceding the newline with a backslash (\):

```
julia> "This is a long \
       line"
"This is a long line"
```

If you want to extract a character from a string, you index into it:

```
julia> str[begin]
'H': ASCII/Unicode U+0048 (category Lu: Letter, uppercase)

julia> str[1]
'H': ASCII/Unicode U+0048 (category Lu: Letter, uppercase)

julia> str[6]
',': ASCII/Unicode U+002C (category Po: Punctuation, other)

julia> str[end]
'\n': ASCII/Unicode U+000A (category Cc: Other, control)
```

Many Julia objects, including strings, can be indexed with integers. The index of the first element (the first character of a string) is returned by `firstindex(str)`, and the index of the last element (character) with `lastindex(str)`. The keywords `begin` and `end` can be used inside an indexing operation as shorthand for the first and last indices, respectively, along the given dimension. String indexing, like most indexing in Julia, is 1-based: `firstindex` always returns 1 for any `AbstractString`. As we will see below, however, `lastindex(str)` is *not* in general the same as `length(str)` for a string, because some Unicode characters can occupy multiple "code units".

You can perform arithmetic and other operations with `end`, just like a normal value:

```
julia> str[end-1]
'.': ASCII/Unicode U+002E (category Po: Punctuation, other)

julia> str[end÷2]
' ': ASCII/Unicode U+0020 (category Zs: Separator, space)
```

Using an index less than `begin` (1) or greater than `end` raises an error:

```
julia> str[begin-1]
ERROR: BoundsError: attempt to access 14-codeunit String at index [0]
[...]

julia> str[end+1]
ERROR: BoundsError: attempt to access 14-codeunit String at index [15]
[...]
```

You can also extract a substring using range indexing:

```
julia> str[4:9]
"lo, wo"
```

Notice that the expressions `str[k]` and `str[k:k]` do not give the same result:

```
julia> str[6]
',': ASCII/Unicode U+002C (category Po: Punctuation, other)

julia> str[6:6]
","
```

The former is a single character value of type `Char`, while the latter is a string value that happens to contain only a single character. In Julia these are very different things.

Range indexing makes a copy of the selected part of the original string. Alternatively, it is possible to create a view into a string using the type `SubString`. More simply, using the `@views` macro on a block of code converts all string slices into substrings. For example:

```
julia> str = "long string"
"long string"

julia> substr = SubString(str, 1, 4)
"long"

julia> typeof(substr)
SubString{String}

julia> @views typeof(str[1:4]) # @views converts slices to SubStrings
SubString{String}
```

Several standard functions like `chop`, `chomp` or `strip` return a `SubString`.

8.3 Unicode and UTF-8

Julia fully supports Unicode characters and strings. As discussed above, in character literals, Unicode code points can be represented using Unicode `\u` and `\U` escape sequences, as well as all the standard C escape sequences. These can likewise be used to write string literals:

```
julia> s = "\u2200 x \u2203 y"
"∀ x ∃ y"
```

Whether these Unicode characters are displayed as escapes or shown as special characters depends on your terminal's locale settings and its support for Unicode. String literals are encoded using the UTF-8 encoding. UTF-8 is a variable-width encoding, meaning that not all characters are encoded in the same number of bytes ("code units"). In UTF-8, ASCII characters — i.e. those with code points less than 0x80 (128) – are encoded as they are in ASCII, using a single byte, while code points 0x80 and above are encoded using multiple bytes — up to four per character.

String indices in Julia refer to code units (= bytes for UTF-8), the fixed-width building blocks that are used to encode arbitrary characters (code points). This means that not every index into a `String` is necessarily a valid index for a character. If you index into a string at such an invalid byte index, an error is thrown:

```
julia> s[1]
'∀': Unicode U+2200 (category Sm: Symbol, math)

julia> s[2]
ERROR: StringIndexError: invalid index [2], valid nearby indices [1]=>'∀', [4]=>' '
Stacktrace:
[...]

julia> s[3]
ERROR: StringIndexError: invalid index [3], valid nearby indices [1]=>'∀', [4]=>' '
Stacktrace:
[...]

julia> s[4]
' ': ASCII/Unicode U+0020 (category Zs: Separator, space)
```

In this case, the character ∀ is a three-byte character, so the indices 2 and 3 are invalid and the next character's index is 4; this next valid index can be computed by `nextind(s,1)`, and the next index after that by `nextind(s,4)` and so on.

Since `end` is always the last valid index into a collection, `end-1` references an invalid byte index if the second-to-last character is multibyte.

```
julia> s[end-1]
' ': ASCII/Unicode U+0020 (category Zs: Separator, space)

julia> s[end-2]
ERROR: StringIndexError: invalid index [9], valid nearby indices [7]=>'∃', [10]=>' '
Stacktrace:
[...]

julia> s[prevind(s, end, 2)]
'∃': Unicode U+2203 (category Sm: Symbol, math)
```

The first case works, because the last character `y` and the space are one-byte characters, whereas `end-2` indexes into the middle of the ∃ multibyte representation. The correct way for this case is using `prevind(s, lastindex(s), 2)` or, if you're using that value to index into `s` you can write `s[prevind(s, end, 2)]` and `end` expands to `lastindex(s)`.

Extraction of a substring using range indexing also expects valid byte indices or an error is thrown:

```
julia> s[1:1]
"∀"

julia> s[1:2]
ERROR: StringIndexError: invalid index [2], valid nearby indices [1]=>'∀', [4]=>' '
Stacktrace:
[...]

julia> s[1:4]
"∀ "
```

Because of variable-length encodings, the number of characters in a string (given by `length(s)`) is not always the same as the last index. If you iterate through the indices 1 through `lastindex(s)` and index into `s`, the sequence of characters returned when errors aren't thrown is the sequence of characters comprising the string `s`. Thus `length(s) <= lastindex(s)`, since each character in a string must have its own index. The following is an inefficient and verbose way to iterate through the characters of `s`:

```
julia> for i = firstindex(s):lastindex(s)
           try
               println(s[i])
           catch
               # ignore the index error
           end
       end
∀

x

∃

y
```

The blank lines actually have spaces on them. Fortunately, the above awkward idiom is unnecessary for iterating through the characters in a string, since you can just use the string as an iterable object, no exception handling required:

```
julia> for c in s
           println(c)
       end
∀

x

∃

y
```

If you need to obtain valid indices for a string, you can use the `nextind` and `prevind` functions to increment/decrement to the next/previous valid index, as mentioned above. You can also use the `eachindex` function to iterate over the valid character indices:

```
julia> collect(eachindex(s))
7-element Vector{Int64}:
  1
  4
  5
  6
  7
 10
 11
```

To access the raw code units (bytes for UTF-8) of the encoding, you can use the `codeunit(s,i)` function, where the index `i` runs consecutively from 1 to `ncodeunits(s)`. The `codeunits(s)` function returns an `AbstractVector{UInt8}` wrapper that lets you access these raw codeunits (bytes) as an array.

Strings in Julia can contain invalid UTF-8 code unit sequences. This convention allows to treat any byte sequence as a `String`. In such situations a rule is that when parsing a sequence of code units from left to right characters are formed by the longest sequence of 8-bit code units that matches the start of one of the following bit patterns (each x can be 0 or 1):

- `0xxxxxxx`;
- `110xxxxx 10xxxxxx`;
- `1110xxxx 10xxxxxx 10xxxxxx`;
- `11110xxx 10xxxxxx 10xxxxxx 10xxxxxx`;
- `10xxxxxx`;
- `11111xxx`.

In particular this means that overlong and too-high code unit sequences and prefixes thereof are treated as a single invalid character rather than multiple invalid characters. This rule may be best explained with an example:

```
julia> s = "\xc0\xa0\xe2\x88\xe2|"
"\xc0\xa0\xe2\x88\xe2|"

julia> foreach(display, s)
'\xc0\xa0': [overlong] ASCII/Unicode U+0020 (category Zs: Separator, space)
'\xe2\x88': Malformed UTF-8 (category Ma: Malformed, bad data)
'\xe2': Malformed UTF-8 (category Ma: Malformed, bad data)
'|': ASCII/Unicode U+007C (category Sm: Symbol, math)

julia> isvalid.(collect(s))
4-element BitArray{1}:
 0
 0
 0
 1

julia> s2 = "\xf7\xbf\xbf\xbf"
```

```
"\U1fffff"

julia> foreach(display, s2)
'\U1fffff': Unicode U+1FFFFF (category In: Invalid, too high)
```

We can see that the first two code units in the string `s` form an overlong encoding of space character. It is invalid, but is accepted in a string as a single character. The next two code units form a valid start of a three-byte UTF-8 sequence. However, the fifth code unit `\xe2` is not its valid continuation. Therefore code units 3 and 4 are also interpreted as malformed characters in this string. Similarly code unit 5 forms a malformed character because `|` is not a valid continuation to it. Finally the string `s2` contains one too high code point.

Julia uses the UTF-8 encoding by default, and support for new encodings can be added by packages. For example, the LegacyStrings.jl package implements `UTF16String` and `UTF32String` types. Additional discussion of other encodings and how to implement support for them is beyond the scope of this document for the time being. For further discussion of UTF-8 encoding issues, see the section below on byte array literals. The `transcode` function is provided to convert data between the various UTF-xx encodings, primarily for working with external data and libraries.

8.4 Concatenation

One of the most common and useful string operations is concatenation:

```
julia> greet = "Hello"
"Hello"

julia> whom = "world"
"world"

julia> string(greet, ", ", whom, ".\n")
"Hello, world.\n"
```

It's important to be aware of potentially dangerous situations such as concatenation of invalid UTF-8 strings. The resulting string may contain different characters than the input strings, and its number of characters may be lower than sum of numbers of characters of the concatenated strings, e.g.:

```
julia> a, b = "\xe2\x88", "\x80"
("\xe2\x88", "\x80")

julia> c = string(a, b)
"∀"

julia> collect.([a, b, c])
3-element Vector{Vector{Char}}:
 ['\xe2\x88']
 ['\x80']
 ['∀']

julia> length.([a, b, c])
3-element Vector{Int64}:
```

```
1
1
1
```

This situation can happen only for invalid UTF-8 strings. For valid UTF-8 strings concatenation preserves all characters in strings and additivity of string lengths.

Julia also provides * for string concatenation:

```
julia> greet * ", " * whom * ".\n"
"Hello, world.\n"
```

While * may seem like a surprising choice to users of languages that provide + for string concatenation, this use of * has precedent in mathematics, particularly in abstract algebra.

In mathematics, + usually denotes a *commutative* operation, where the order of the operands does not matter. An example of this is matrix addition, where `A + B == B + A` for any matrices `A` and `B` that have the same shape. In contrast, * typically denotes a *noncommutative* operation, where the order of the operands *does* matter. An example of this is matrix multiplication, where in general `A * B != B * A`. As with matrix multiplication, string concatenation is noncommutative: `greet * whom != whom * greet`. As such, * is a more natural choice for an infix string concatenation operator, consistent with common mathematical use.

More precisely, the set of all finite-length strings S together with the string concatenation operator * forms a free monoid (S, *). The identity element of this set is the empty string, `""`. Whenever a free monoid is not commutative, the operation is typically represented as `\cdot`, *, or a similar symbol, rather than +, which as stated usually implies commutativity.

8.5 Interpolation

Constructing strings using concatenation can become a bit cumbersome, however. To reduce the need for these verbose calls to `string` or repeated multiplications, Julia allows interpolation into string literals using `$`, as in Perl:

```
julia> greet = "Hello"; whom = "world";

julia> "$greet, $whom.\n"
"Hello, world.\n"
```

This is more readable and convenient and equivalent to the above string concatenation – the system rewrites this apparent single string literal into the call `string(greet, ", ", whom, ".\n")`.

The shortest complete expression after the `$` is taken as the expression whose value is to be interpolated into the string. Thus, you can interpolate any expression into a string using parentheses:

```
julia> "1 + 2 = $(1 + 2)"
"1 + 2 = 3"
```

Both concatenation and string interpolation call `string` to convert objects into string form. However, `string` actually just returns the output of `print`, so new types should add methods to `print` or `show` instead of `string`.

Most non-`AbstractString` objects are converted to strings closely corresponding to how they are entered as literal expressions:

```
julia> v = [1,2,3]
3-element Vector{Int64}:
 1
 2
 3

julia> "v: $v"
"v: [1, 2, 3]"
```

`string` is the identity for `AbstractString` and `AbstractChar` values, so these are interpolated into strings as themselves, unquoted and unescaped:

```
julia> c = 'x'
'x': ASCII/Unicode U+0078 (category Ll: Letter, lowercase)

julia> "hi, $c"
"hi, x"
```

To include a literal `$` in a string literal, escape it with a backslash:

```
julia> print("I have \$100 in my account.\n")
I have $100 in my account.
```

8.6 Triple-Quoted String Literals

When strings are created using triple-quotes (`"""..."""`) they have some special behavior that can be useful for creating longer blocks of text.

First, triple-quoted strings are also dedented to the level of the least-indented line. This is useful for defining strings within code that is indented. For example:

```
julia> str = """
           Hello,
           world.
         """
"  Hello,\n  world.\n"
```

In this case the final (empty) line before the closing `"""` sets the indentation level.

The dedentation level is determined as the longest common starting sequence of spaces or tabs in all lines, excluding the line following the opening `"""` and lines containing only spaces or tabs (the line containing the closing `"""` is always included). Then for all lines, excluding the text following the opening `"""`, the common starting sequence is removed (including lines containing only spaces and tabs if they start with this sequence), e.g.:

```
julia> """    This
         is
           a test"""
"    This\nis\n  a test"
```

Next, if the opening """ is followed by a newline, the newline is stripped from the resulting string.

```
"""hello"""
```

is equivalent to

```
"""
hello"""
```

but

```
"""

hello"""
```

will contain a literal newline at the beginning.

Stripping of the newline is performed after the dedentation. For example:

```
julia> """
         Hello,
         world."""
"Hello,\nworld."
```

If the newline is removed using a backslash, dedentation will be respected as well:

```
julia> """
         Averylong\
         word"""
"Averylongword"
```

Trailing whitespace is left unaltered.

Triple-quoted string literals can contain " characters without escaping.

Note that line breaks in literal strings, whether single- or triple-quoted, result in a newline (LF) character `\n` in the string, even if your editor uses a carriage return `\r` (CR) or CRLF combination to end lines. To include a CR in a string, use an explicit escape `\r`; for example, you can enter the literal string `"a CRLF line ending\r\n"`.

8.7 Common Operations

You can lexicographically compare strings using the standard comparison operators:

```
julia> "abracadabra" < "xylophone"
true

julia> "abracadabra" == "xylophone"
false

julia> "Hello, world." != "Goodbye, world."
true

julia> "1 + 2 = 3" == "1 + 2 = $(1 + 2)"
true
```

You can search for the index of a particular character using the `findfirst` and `findlast` functions:

```
julia> findfirst('o', "xylophone")
4

julia> findlast('o', "xylophone")
7

julia> findfirst('z', "xylophone")
```

You can start the search for a character at a given offset by using the functions `findnext` and `findprev`:

```
julia> findnext('o', "xylophone", 1)
4

julia> findnext('o', "xylophone", 5)
7

julia> findprev('o', "xylophone", 5)
4

julia> findnext('o', "xylophone", 8)
```

You can use the `occursin` function to check if a substring is found within a string:

```
julia> occursin("world", "Hello, world.")
true

julia> occursin("o", "Xylophon")
true

julia> occursin("a", "Xylophon")
false

julia> occursin('o', "Xylophon")
true
```

The last example shows that `occursin` can also look for a character literal.

Two other handy string functions are `repeat` and `join`:

```
julia> repeat(".:Z:.", 10)
".:Z:..:Z:..:Z:..:Z:..:Z:..:Z:..:Z:..:Z:..:Z:.."

julia> join(["apples", "bananas", "pineapples"], ", ", " and ")
"apples, bananas and pineapples"
```

Some other useful functions include:

- `firstindex(str)` gives the minimal (byte) index that can be used to index into `str` (always 1 for strings, not necessarily true for other containers).
- `lastindex(str)` gives the maximal (byte) index that can be used to index into `str`.
- `length(str)` the number of characters in `str`.
- `length(str, i, j)` the number of valid character indices in `str` from `i` to `j`.
- `ncodeunits(str)` number of code units in a string.
- `codeunit(str, i)` gives the code unit value in the string `str` at index `i`.
- `thisind(str, i)` given an arbitrary index into a string find the first index of the character into which the index points.
- `nextind(str, i, n=1)` find the start of the nth character starting after index `i`.
- `prevind(str, i, n=1)` find the start of the nth character starting before index `i`.

8.8 Non-Standard String Literals

There are situations when you want to construct a string or use string semantics, but the behavior of the standard string construct is not quite what is needed. For these kinds of situations, Julia provides non-standard string literals. A non-standard string literal looks like a regular double-quoted string literal, but is immediately prefixed by an identifier, and may behave differently from a normal string literal.

Regular expressions, byte array literals, and version number literals, as described below, are some examples of non-standard string literals. Users and packages may also define new non-standard string literals. Further documentation is given in the Metaprogramming section.

8.9 Regular Expressions

Sometimes you are not looking for an exact string, but a particular *pattern*. For example, suppose you are trying to extract a single date from a large text file. You don't know what that date is (that's why you are searching for it), but you do know it will look something like `YYYY-MM-DD`. Regular expressions allow you to specify these patterns and search for them.

Julia uses version 2 of Perl-compatible regular expressions (regexes), as provided by the PCRE library (see the PCRE2 syntax description for more details). Regular expressions are related to strings in two ways:

the obvious connection is that regular expressions are used to find regular patterns in strings; the other connection is that regular expressions are themselves input as strings, which are parsed into a state machine that can be used to efficiently search for patterns in strings. In Julia, regular expressions are input using non-standard string literals prefixed with various identifiers beginning with `r`. The most basic regular expression literal without any options turned on just uses `r"..."`:

```
julia> re = r"^\s*(?:#|$)"
r"^\s*(?:#|$)"

julia> typeof(re)
Regex
```

To check if a regex matches a string, use `occursin`:

```
julia> occursin(r"^\s*(?:#|$)", "not a comment")
false

julia> occursin(r"^\s*(?:#|$)", "# a comment")
true
```

As one can see here, `occursin` simply returns true or false, indicating whether a match for the given regex occurs in the string. Commonly, however, one wants to know not just whether a string matched, but also *how* it matched. To capture this information about a match, use the `match` function instead:

```
julia> match(r"^\s*(?:#|$)", "not a comment")

julia> match(r"^\s*(?:#|$)", "# a comment")
RegexMatch("#")
```

If the regular expression does not match the given string, `match` returns `nothing` – a special value that does not print anything at the interactive prompt. Other than not printing, it is a completely normal value and you can test for it programmatically:

```
m = match(r"^\s*(?:#|$)", line)
if m === nothing
    println("not a comment")
else
    println("blank or comment")
end
```

If a regular expression does match, the value returned by `match` is a `RegexMatch` object. These objects record how the expression matches, including the substring that the pattern matches and any captured substrings, if there are any. This example only captures the portion of the substring that matches, but perhaps we want to capture any non-blank text after the comment character. We could do the following:

```
julia> m = match(r"^\s*(?:#\s*(.*?)\s*$)", "# a comment ")
RegexMatch("# a comment ", 1="a comment")
```

When calling `match`, you have the option to specify an index at which to start the search. For example:

```
julia> m = match(r"[0-9]","aaaa1aaaa2aaaa3",1)
RegexMatch("1")

julia> m = match(r"[0-9]","aaaa1aaaa2aaaa3",6)
RegexMatch("2")

julia> m = match(r"[0-9]","aaaa1aaaa2aaaa3",11)
RegexMatch("3")
```

You can extract the following info from a `RegexMatch` object:

- the entire substring matched: `m.match`
- the captured substrings as an array of strings: `m.captures`
- the offset at which the whole match begins: `m.offset`
- the offsets of the captured substrings as a vector: `m.offsets`

For when a capture doesn't match, instead of a substring, `m.captures` contains `nothing` in that position, and `m.offsets` has a zero offset (recall that indices in Julia are 1-based, so a zero offset into a string is invalid). Here is a pair of somewhat contrived examples:

```
julia> m = match(r"(a|b)(c)?(d)", "acd")
RegexMatch("acd", 1="a", 2="c", 3="d")

julia> m.match
"acd"

julia> m.captures
3-element Vector{Union{Nothing, SubString{String}}}:
 "a"
 "c"
 "d"

julia> m.offset
1

julia> m.offsets
3-element Vector{Int64}:
 1
 2
 3

julia> m = match(r"(a|b)(c)?(d)", "ad")
RegexMatch("ad", 1="a", 2=nothing, 3="d")

julia> m.match
"ad"
```

```
julia> m.captures
3-element Vector{Union{Nothing, SubString{String}}}:
 "a"
 nothing
 "d"

julia> m.offset
1

julia> m.offsets
3-element Vector{Int64}:
 1
 0
 2
```

It is convenient to have captures returned as an array so that one can use destructuring syntax to bind them to local variables. As a convenience, the `RegexMatch` object implements iterator methods that pass through to the `captures` field, so you can destructure the match object directly:

```
julia> first, second, third = m; first
"a"
```

Captures can also be accessed by indexing the `RegexMatch` object with the number or name of the capture group:

```
julia> m=match(r"(?<hour>\d+):(?<minute>\d+)","12:45")
RegexMatch("12:45", hour="12", minute="45")

julia> m[:minute]
"45"

julia> m[2]
"45"
```

Captures can be referenced in a substitution string when using `replace` by using `\n` to refer to the nth capture group and prefixing the substitution string with `s`. Capture group 0 refers to the entire match object. Named capture groups can be referenced in the substitution with `\g<groupname>`. For example:

```
julia> replace("first second", r"(\w+) (?<agroup>\w+)" => s"\g<agroup> \1")
"second first"
```

Numbered capture groups can also be referenced as `\g<n>` for disambiguation, as in:

```
julia> replace("a", r"." => s"\g<0>1")
"a1"
```

You can modify the behavior of regular expressions by some combination of the flags `i`, `m`, `s`, and `x` after the closing double quote mark. These flags have the same meaning as they do in Perl, as explained in this excerpt from the perlre manpage:

```
i   Do case-insensitive pattern matching.

    If locale matching rules are in effect, the case map is taken
    from the current locale for code points less than 255, and
    from Unicode rules for larger code points. However, matches
    that would cross the Unicode rules/non-Unicode rules boundary
    (ords 255/256) will not succeed.

m   Treat string as multiple lines. That is, change "^" and "$"
    from matching the start or end of the string to matching the
    start or end of any line anywhere within the string.

s   Treat string as single line. That is, change "." to match any
    character whatsoever, even a newline, which normally it would
    not match.

    Used together, as r""ms, they let the "." match any character
    whatsoever, while still allowing "^" and "$" to match,
    respectively, just after and just before newlines within the
    string.

x   Tells the regular expression parser to ignore most whitespace
    that is neither backslashed nor within a character class. You
    can use this to break up your regular expression into
    (slightly) more readable parts. The '#' character is also
    treated as a metacharacter introducing a comment, just as in
    ordinary code.
```

For example, the following regex has all three flags turned on:

```
julia> r"a+.*b+.*d$"ism
r"a+.*b+.*d$"ims

julia> match(r"a+.*b+.*d$"ism, "Goodbye,\nOh, angry,\nBad world\n")
RegexMatch("angry,\nBad world")
```

The `r"..."` literal is constructed without interpolation and unescaping (except for quotation mark `"` which still has to be escaped). Here is an example showing the difference from standard string literals:

```
julia> x = 10
10

julia> r"$x"
r"$x"

julia> "$x"
"10"

julia> r"\x"
r"\x"
```

```
julia> "\x"
ERROR: ParseError:
# Error @ none:1:2
"\x"
#␣ ── invalid hex escape sequence
```

Triple-quoted regex strings, of the form `r"""..."""`, are also supported (and may be convenient for regular expressions containing quotation marks or newlines).

The `Regex()` constructor may be used to create a valid regex string programmatically. This permits using the contents of string variables and other string operations when constructing the regex string. Any of the regex codes above can be used within the single string argument to `Regex()`. Here are some examples:

```
julia> using Dates

julia> d = Date(1962,7,10)
1962-07-10

julia> regex_d = Regex("Day " * string(day(d)))
r"Day 10"

julia> match(regex_d, "It happened on Day 10")
RegexMatch("Day 10")

julia> name = "Jon"
"Jon"

julia> regex_name = Regex("[\"( ]\\Q$name\\E[\") ]")  # interpolate value of name
r"[\"( ]\QJon\E[\") ]"

julia> match(regex_name, " Jon ")
RegexMatch(" Jon ")

julia> match(regex_name, "[Jon]") === nothing
true
```

Note the use of the `\Q...\E` escape sequence. All characters between the `\Q` and the `\E` are interpreted as literal characters. This is convenient for matching characters that would otherwise be regex metacharacters. However, caution is needed when using this feature together with string interpolation, since the interpolated string might itself contain the `\E` sequence, unexpectedly terminating literal matching. User inputs need to be sanitized before inclusion in a regex.

8.10 Byte Array Literals

Another useful non-standard string literal is the byte-array string literal: `b"..."`. This form lets you use string notation to express read only literal byte arrays – i.e. arrays of `UInt8` values. The type of those objects is `CodeUnits{UInt8, String}`. The rules for byte array literals are the following:

- ASCII characters and ASCII escapes produce a single byte.
- `\x` and octal escape sequences produce the *byte* corresponding to the escape value.

- Unicode escape sequences produce a sequence of bytes encoding that code point in UTF-8.

There is some overlap between these rules since the behavior of \x and octal escapes less than 0x80 (128) are covered by both of the first two rules, but here these rules agree. Together, these rules allow one to easily use ASCII characters, arbitrary byte values, and UTF-8 sequences to produce arrays of bytes. Here is an example using all three:

```
julia> b"DATA\xff\u2200"
8-element Base.CodeUnits{UInt8, String}:
 0x44
 0x41
 0x54
 0x41
 0xff
 0xe2
 0x88
 0x80
```

The ASCII string "DATA" corresponds to the bytes 68, 65, 84, 65. `\xff` produces the single byte 255. The Unicode escape `\u2200` is encoded in UTF-8 as the three bytes 226, 136, 128. Note that the resulting byte array does not correspond to a valid UTF-8 string:

```
julia> isvalid("DATA\xff\u2200")
false
```

As it was mentioned `CodeUnits{UInt8, String}` type behaves like read only array of `UInt8` and if you need a standard vector you can convert it using `Vector{UInt8}`:

```
julia> x = b"123"
3-element Base.CodeUnits{UInt8, String}:
 0x31
 0x32
 0x33

julia> x[1]
0x31

julia> x[1] = 0x32
ERROR: CanonicalIndexError: setindex! not defined for Base.CodeUnits{UInt8, String}
[...]

julia> Vector{UInt8}(x)
3-element Vector{UInt8}:
 0x31
 0x32
 0x33
```

Also observe the significant distinction between `\xff` and `\uff`: the former escape sequence encodes the *byte 255*, whereas the latter escape sequence represents the *code point 255*, which is encoded as two bytes in UTF-8:

```
julia> b"\xff"
1-element Base.CodeUnits{UInt8, String}:
 0xff

julia> b"\uff"
2-element Base.CodeUnits{UInt8, String}:
 0xc3
 0xbf
```

Character literals use the same behavior.

For code points less than `\u80`, it happens that the UTF-8 encoding of each code point is just the single byte produced by the corresponding `\x` escape, so the distinction can safely be ignored. For the escapes `\x80` through `\xff` as compared to `\u80` through `\uff`, however, there is a major difference: the former escapes all encode single bytes, which – unless followed by very specific continuation bytes – do not form valid UTF-8 data, whereas the latter escapes all represent Unicode code points with two-byte encodings.

If this is all extremely confusing, try reading "The Absolute Minimum Every Software Developer Absolutely, Positively Must Know About Unicode and Character Sets". It's an excellent introduction to Unicode and UTF-8, and may help alleviate some confusion regarding the matter.

8.11 Version Number Literals

Version numbers can easily be expressed with non-standard string literals of the form `v"..."`. Version number literals create `VersionNumber` objects which follow the specifications of semantic versioning 2.0.0-rc2, and therefore are composed of major, minor and patch numeric values, followed by pre-release and build alphanumeric annotations. For example, `v"0.2.1-rc1+win64"` is broken into major version `0`, minor version 2, patch version 1, pre-release `rc1` and build `win64`. When entering a version literal, everything except the major version number is optional, therefore e.g. `v"0.2"` is equivalent to `v"0.2.0"` (with empty pre-release/build annotations), `v"2"` is equivalent to `v"2.0.0"`, and so on.

`VersionNumber` objects are mostly useful to easily and correctly compare two (or more) versions. For example, the constant `VERSION` holds Julia version number as a `VersionNumber` object, and therefore one can define some version-specific behavior using simple statements as:

```
if v"0.2" <= VERSION < v"0.3-"
    # do something specific to 0.2 release series
end
```

Note that in the above example the non-standard version number `v"0.3-"` is used, with a trailing `-`: this notation is a Julia extension of the standard, and it's used to indicate a version which is lower than any `0.3` release, including all of its pre-releases. So in the above example the code would only run with stable `0.2` versions, and exclude such versions as `v"0.3.0-rc1"`. In order to also allow for unstable (i.e. pre-release) `0.2` versions, the lower bound check should be modified like this: `v"0.2-" <= VERSION`.

Another non-standard version specification extension allows one to use a trailing `+` to express an upper limit on build versions, e.g. `VERSION > v"0.2-rc1+"` can be used to mean any version above `0.2-rc1` and any of its builds: it will return `false` for version `v"0.2-rc1+win64"` and `true` for `v"0.2-rc2"`.

It is good practice to use such special versions in comparisons (particularly, the trailing `-` should always be used on upper bounds unless there's a good reason not to), but they must not be used as the actual version number of anything, as they are invalid in the semantic versioning scheme.

Besides being used for the VERSION constant, VersionNumber objects are widely used in the Pkg module, to specify packages versions and their dependencies.

8.12 Raw String Literals

Raw strings without interpolation or unescaping can be expressed with non-standard string literals of the form raw"...". Raw string literals create ordinary String objects which contain the enclosed contents exactly as entered with no interpolation or unescaping. This is useful for strings which contain code or markup in other languages which use $ or \ as special characters.

The exception is that quotation marks still must be escaped, e.g. raw"\"" is equivalent to "\"". To make it possible to express all strings, backslashes then also must be escaped, but only when appearing right before a quote character:

```
julia> println(raw"\\ \\\"")
\\ \"
```

Notice that the first two backslashes appear verbatim in the output, since they do not precede a quote character. However, the next backslash character escapes the backslash that follows it, and the last backslash escapes a quote, since these backslashes appear before a quote.

8.13 Annotated Strings

Note

The API for AnnotatedStrings is considered experimental and is subject to change between Julia versions.

It is sometimes useful to be able to hold metadata relating to regions of a string. A AnnotatedString wraps another string and allows for regions of it to be annotated with labelled values (:label => value). All generic string operations are applied to the underlying string. However, when possible, styling information is preserved. This means you can manipulate a AnnotatedString —taking substrings, padding them, concatenating them with other strings— and the metadata annotations will "come along for the ride".

This string type is fundamental to the StyledStrings stdlib, which uses :face-labelled annotations to hold styling information.

When concatenating a AnnotatedString, take care to use annotatedstring instead of string if you want to keep the string annotations.

```
julia> str = Base.AnnotatedString("hello there",
               [(1:5, :word, :greeting), (7:11, :label, 1)])
"hello there"

julia> length(str)
11

julia> lpad(str, 14)
"   hello there"
```

```
julia> typeof(lpad(str, 7))
Base.AnnotatedString{String}

julia> str2 = Base.AnnotatedString(" julia", [(2:6, :face, :magenta)])
" julia"

julia> Base.annotatedstring(str, str2)
"hello there julia"

julia> str * str2 == Base.annotatedstring(str, str2) # *-concatenation still works
true
```

The annotations of a `AnnotatedString` can be accessed and modified via the `annotations` and `annotate!` functions.

Chapter 9

Functions

In Julia, a function is an object that maps a tuple of argument values to a return value. Julia functions are not pure mathematical functions, because they can alter and be affected by the global state of the program. The basic syntax for defining functions in Julia is:

```
julia> function f(x, y)
           x + y
       end
f (generic function with 1 method)
```

This function accepts two arguments x and y and returns the value of the last expression evaluated, which is `x + y`.

There is a second, more terse syntax for defining a function in Julia. The traditional function declaration syntax demonstrated above is equivalent to the following compact "assignment form":

```
julia> f(x, y) = x + y
f (generic function with 1 method)
```

In the assignment form, the body of the function must be a single expression, although it can be a compound expression (see Compound Expressions). Short, simple function definitions are common in Julia. The short function syntax is accordingly quite idiomatic, considerably reducing both typing and visual noise.

A function is called using the traditional parenthesis syntax:

```
julia> f(2, 3)
5
```

Without parentheses, the expression `f` refers to the function object, and can be passed around like any other value:

```
julia> g = f;

julia> g(2, 3)
5
```

As with variables, Unicode can also be used for function names:

```
julia> ∑(x, y) = x + y
∑ (generic function with 1 method)

julia> ∑(2, 3)
5
```

9.1 Argument Passing Behavior

Julia function arguments follow a convention sometimes called "pass-by-sharing", which means that values are not copied when they are passed to functions. Function arguments themselves act as new variable *bindings* (new "names" that can refer to values), much like assignments `argument_name = argument_value`, so that the objects they refer to are identical to the passed values. Modifications to mutable values (such as `Arrays`) made within a function will be visible to the caller. (This is the same behavior found in Scheme, most Lisps, Python, Ruby and Perl, among other dynamic languages.)

For example, in the function

```
function f(x, y)
    x[1] = 42    # mutates x
    y = 7 + y    # new binding for y, no mutation
    return y
end
```

The statement `x[1] = 42` *mutates* the object `x`, and hence this change *will* be visible in the array passed by the caller for this argument. On the other hand, the assignment `y = 7 + y` changes the *binding* ("name") `y` to refer to a new value `7 + y`, rather than mutating the *original* object referred to by `y`, and hence does *not* change the corresponding argument passed by the caller. This can be seen if we call `f(x, y)`:

```
julia> a = [4, 5, 6]
3-element Vector{Int64}:
 4
 5
 6

julia> b = 3
3

julia> f(a, b) # returns 7 + b == 10
10

julia> a  # a[1] is changed to 42 by f
3-element Vector{Int64}:
 42
  5
  6

julia> b  # not changed
3
```

As a common convention in Julia (not a syntactic requirement), such a function would typically be named `f!(x, y)` rather than `f(x, y)`, as a visual reminder at the call site that at least one of the arguments (often the first one) is being mutated.

Shared memory between arguments

The behavior of a mutating function can be unexpected when a mutated argument shares memory with another argument, a situation known as aliasing (e.g. when one is a view of the other). Unless the function docstring explicitly indicates that aliasing produces the expected result, it is the responsibility of the caller to ensure proper behavior on such inputs.

9.2 Argument-type declarations

You can declare the types of function arguments by appending `::TypeName` to the argument name, as usual for Type Declarations in Julia. For example, the following function computes Fibonacci numbers recursively:

```
fib(n::Integer) = n ≤ 2 ? one(n) : fib(n-1) + fib(n-2)
```

and the `::Integer` specification means that it will only be callable when `n` is a subtype of the abstract `Integer` type.

Argument-type declarations **normally have no impact on performance**: regardless of what argument types (if any) are declared, Julia compiles a specialized version of the function for the actual argument types passed by the caller. For example, calling `fib(1)` will trigger the compilation of specialized version of `fib` optimized specifically for `Int` arguments, which is then re-used if `fib(7)` or `fib(15)` are called. (There are rare exceptions when an argument-type declaration can trigger additional compiler specializations; see: Be aware of when Julia avoids specializing.) The most common reasons to declare argument types in Julia are, instead:

- **Dispatch:** As explained in Methods, you can have different versions ("methods") of a function for different argument types, in which case the argument types are used to determine which implementation is called for which arguments. For example, you might implement a completely different algorithm `fib(x::Number) = ...` that works for any `Number` type by using Binet's formula to extend it to non-integer values.
- **Correctness:** Type declarations can be useful if your function only returns correct results for certain argument types. For example, if we omitted argument types and wrote `fib(n) = n ≤ 2 ? one(n) : fib(n-1) + fib(n-2)`, then `fib(1.5)` would silently give us the nonsensical answer `1.0`.
- **Clarity:** Type declarations can serve as a form of documentation about the expected arguments.

However, it is a **common mistake to overly restrict the argument types**, which can unnecessarily limit the applicability of the function and prevent it from being re-used in circumstances you did not anticipate. For example, the `fib(n::Integer)` function above works equally well for `Int` arguments (machine integers) and `BigInt` arbitrary-precision integers (see BigFloats and BigInts), which is especially useful because Fibonacci numbers grow exponentially rapidly and will quickly overflow any fixed-precision type like `Int` (see Overflow behavior). If we had declared our function as `fib(n::Int)`, however, the application to `BigInt` would have been prevented for no reason. In general, you should use the most general applicable abstract types for arguments, and **when in doubt, omit the argument types**. You can always

add argument-type specifications later if they become necessary, and you don't sacrifice performance or functionality by omitting them.

9.3 The `return` Keyword

The value returned by a function is the value of the last expression evaluated, which, by default, is the last expression in the body of the function definition. In the example function, `f`, from the previous section this is the value of the expression `x + y`. As an alternative, as in many other languages, the `return` keyword causes a function to return immediately, providing an expression whose value is returned:

```
function g(x, y)
    return x * y
    x + y
end
```

Since function definitions can be entered into interactive sessions, it is easy to compare these definitions:

```
julia> f(x, y) = x + y
f (generic function with 1 method)

julia> function g(x, y)
           return x * y
           x + y
       end
g (generic function with 1 method)

julia> f(2, 3)
5

julia> g(2, 3)
6
```

Of course, in a purely linear function body like `g`, the usage of `return` is pointless since the expression `x + y` is never evaluated and we could simply make `x * y` the last expression in the function and omit the `return`. In conjunction with other control flow, however, `return` is of real use. Here, for example, is a function that computes the hypotenuse length of a right triangle with sides of length `x` and `y`, avoiding overflow:

```
julia> function hypot(x, y)
           x = abs(x)
           y = abs(y)
           if x > y
               r = y/x
               return x*sqrt(1 + r*r)
           end
           if y == 0
               return float(x)
           end
           r = x/y
           return y*sqrt(1 + r*r)
```

```
      end
hypot (generic function with 1 method)

julia> hypot(3, 4)
5.0
```

There are three possible points of return from this function, returning the values of three different expressions, depending on the values of x and y. The `return` on the last line could be omitted since it is the last expression.

Return type

A return type can be specified in the function declaration using the `::` operator. This converts the return value to the specified type.

```
julia> function g(x, y)::Int8
           return x * y
       end;

julia> typeof(g(1, 2))
Int8
```

This function will always return an `Int8` regardless of the types of x and y. See Type Declarations for more on return types.

Return type declarations are **rarely used** in Julia: in general, you should instead write "type-stable" functions in which Julia's compiler can automatically infer the return type. For more information, see the Performance Tips chapter.

Returning nothing

For functions that do not need to return a value (functions used only for some side effects), the Julia convention is to return the value `nothing`:

```
function printx(x)
    println("x = $x")
    return nothing
end
```

This is a *convention* in the sense that `nothing` is not a Julia keyword but only a singleton object of type `Nothing`. Also, you may notice that the `printx` function example above is contrived, because `println` already returns `nothing`, so that the `return` line is redundant.

There are two possible shortened forms for the `return nothing` expression. On the one hand, the `return` keyword implicitly returns `nothing`, so it can be used alone. On the other hand, since functions implicitly return their last expression evaluated, `nothing` can be used alone when it's the last expression. The preference for the expression `return nothing` as opposed to `return` or `nothing` alone is a matter of coding style.

9.4 Operators Are Functions

In Julia, most operators are just functions with support for special syntax. (The exceptions are operators with special evaluation semantics like && and ||. These operators cannot be functions since Short-Circuit Evaluation requires that their operands are not evaluated before evaluation of the operator.) Accordingly, you can also apply them using parenthesized argument lists, just as you would any other function:

```
julia> 1 + 2 + 3
6

julia> +(1, 2, 3)
6
```

The infix form is exactly equivalent to the function application form – in fact the former is parsed to produce the function call internally. This also means that you can assign and pass around operators such as + and * just like you would with other function values:

```
julia> f = +;

julia> f(1, 2, 3)
6
```

Under the name f, the function does not support infix notation, however.

9.5 Operators With Special Names

A few special expressions correspond to calls to functions with non-obvious names. These are:

Expression	Calls
[A B C ...]	hcat
[A; B; C; ...]	vcat
[A B; C D; ...]	hvcat
[A; B;; C; D;; ...]	hvncat
A'	adjoint
A[i]	getindex
A[i] = x	setindex!
A.n	getproperty
A.n = x	setproperty!

Note that expressions similar to [A; B;; C; D;; ...] but with more than two consecutive ; also correspond to hvncat calls.

9.6 Anonymous Functions

Functions in Julia are first-class objects: they can be assigned to variables, and called using the standard function call syntax from the variable they have been assigned to. They can be used as arguments, and they can be returned as values. They can also be created anonymously, without being given a name, using either of these syntaxes:

```
julia> x -> x^2 + 2x - 1
#2 (generic function with 1 method)

julia> function (x)
           x^2 + 2x - 1
       end
#5 (generic function with 1 method)
```

Each statement creates a function taking one argument x and returning the value of the polynomial x^2 + 2x - 1 at that value. Notice that the result is a generic function, but with a compiler-generated name based on consecutive numbering.

The primary use for anonymous functions is passing them to functions which take other functions as arguments. A classic example is `map`, which applies a function to each value of an array and returns a new array containing the resulting values:

```
julia> map(round, [1.2, 3.5, 1.7])
3-element Vector{Float64}:
 1.0
 4.0
 2.0
```

This is fine if a named function effecting the transform already exists to pass as the first argument to `map`. Often, however, a ready-to-use, named function does not exist. In these situations, the anonymous function construct allows easy creation of a single-use function object without needing a name:

```
julia> map(x -> x^2 + 2x - 1, [1, 3, -1])
3-element Vector{Int64}:
  2
 14
 -2
```

An anonymous function accepting multiple arguments can be written using the syntax `(x,y,z)->2x+y-z`.

Argument-type declarations for anonymous functions work as for named functions, for example `x::Integer->2x`. The return type of an anonymous function cannot be specified.

A zero-argument anonymous function can be written as `()->2+2`. The idea of a function with no arguments may seem strange, but is useful in cases where a result cannot (or should not) be precomputed. For example, Julia has a zero-argument `time` function that returns the current time in seconds, and thus `seconds = ()->round(Int, time())` is an anonymous function that returns this time rounded to the nearest integer assigned to the variable `seconds`. Each time this anonymous function is called as `seconds()` the current time will be calculated and returned.

9.7 Tuples

Julia has a built-in data structure called a *tuple* that is closely related to function arguments and return values. A tuple is a fixed-length container that can hold any values, but cannot be modified (it is *immutable*). Tuples are constructed with commas and parentheses, and can be accessed via indexing:

```
julia> (1, 1+1)
(1, 2)

julia> (1,)
(1,)

julia> x = (0.0, "hello", 6*7)
(0.0, "hello", 42)

julia> x[2]
"hello"
```

Notice that a length-1 tuple must be written with a comma, `(1,)`, since `(1)` would just be a parenthesized value. `()` represents the empty (length-0) tuple.

9.8 Named Tuples

The components of tuples can optionally be named, in which case a *named tuple* is constructed:

```
julia> x = (a=2, b=1+2)
(a = 2, b = 3)

julia> x[1]
2

julia> x.a
2
```

The fields of named tuples can be accessed by name using dot syntax (`x.a`) in addition to the regular indexing syntax (`x[1]` or `x[:a]`).

9.9 Destructuring Assignment and Multiple Return Values

A comma-separated list of variables (optionally wrapped in parentheses) can appear on the left side of an assignment: the value on the right side is *destructured* by iterating over and assigning to each variable in turn:

```
julia> (a, b, c) = 1:3
1:3

julia> b
2
```

The value on the right should be an iterator (see Iteration interface) at least as long as the number of variables on the left (any excess elements of the iterator are ignored).

This can be used to return multiple values from functions by returning a tuple or other iterable value. For example, the following function returns two values:

```
julia> function foo(a, b)
           a+b, a*b
       end
foo (generic function with 1 method)
```

If you call it in an interactive session without assigning the return value anywhere, you will see the tuple returned:

```
julia> foo(2, 3)
(5, 6)
```

Destructuring assignment extracts each value into a variable:

```
julia> x, y = foo(2, 3)
(5, 6)

julia> x
5

julia> y
6
```

Another common use is for swapping variables:

```
julia> y, x = x, y
(5, 6)

julia> x
6

julia> y
5
```

If only a subset of the elements of the iterator are required, a common convention is to assign ignored elements to a variable consisting of only underscores _ (which is an otherwise invalid variable name, see Allowed Variable Names):

```
julia> _, _, _, d = 1:10
1:10

julia> d
4
```

Other valid left-hand side expressions can be used as elements of the assignment list, which will call `setindex!` or `setproperty!`, or recursively destructure individual elements of the iterator:

```
julia> X = zeros(3);

julia> X[1], (a, b) = (1, (2, 3))
(1, (2, 3))

julia> X
3-element Vector{Float64}:
 1.0
 0.0
 0.0

julia> a
2

julia> b
3
```

Julia 1.6

`...` with assignment requires Julia 1.6

If the last symbol in the assignment list is suffixed by `...` (known as *slurping*), then it will be assigned a collection or lazy iterator of the remaining elements of the right-hand side iterator:

```
julia> a, b... = "hello"
"hello"

julia> a
'h': ASCII/Unicode U+0068 (category Ll: Letter, lowercase)

julia> b
"ello"

julia> a, b... = Iterators.map(abs2, 1:4)
Base.Generator{UnitRange{Int64}, typeof(abs2)}(abs2, 1:4)

julia> a
1

julia> b
Base.Iterators.Rest{Base.Generator{UnitRange{Int64}, typeof(abs2)},
↪  Int64}(Base.Generator{UnitRange{Int64}, typeof(abs2)}(abs2, 1:4), 1)
```

See `Base.rest` for details on the precise handling and customization for specific iterators.

Julia 1.9

`...` in non-final position of an assignment requires Julia 1.9

Slurping in assignments can also occur in any other position. As opposed to slurping the end of a collection however, this will always be eager.

```
julia> a, b..., c = 1:5
1:5

julia> a
1

julia> b
3-element Vector{Int64}:
 2
 3
 4

julia> c
5

julia> front..., tail = "Hi!"
"Hi!"

julia> front
"Hi"

julia> tail
'!': ASCII/Unicode U+0021 (category Po: Punctuation, other)
```

This is implemented in terms of the function `Base.split_rest`.

Note that for variadic function definitions, slurping is still only allowed in final position. This does not apply to single argument destructuring though, as that does not affect method dispatch:

```
julia> f(x..., y) = x
ERROR: syntax: invalid "..." on non-final argument
Stacktrace:
[...]

julia> f((x..., y)) = x
f (generic function with 1 method)

julia> f((1, 2, 3))
(1, 2)
```

9.10 Property destructuring

Instead of destructuring based on iteration, the right side of assignments can also be destructured using property names. This follows the syntax for NamedTuples, and works by assigning to each variable on the left a property of the right side of the assignment with the same name using `getproperty`:

```
julia> (; b, a) = (a=1, b=2, c=3)
(a = 1, b = 2, c = 3)

julia> a
1
```

```
julia> b
2
```

9.11 Argument destructuring

The destructuring feature can also be used within a function argument. If a function argument name is written as a tuple (e.g. `(x, y)`) instead of just a symbol, then an assignment `(x, y) = argument` will be inserted for you:

```
julia> minmax(x, y) = (y < x) ? (y, x) : (x, y)
minmax (generic function with 1 method)

julia> gap((min, max)) = max - min
gap (generic function with 1 method)

julia> gap(minmax(10, 2))
8
```

Notice the extra set of parentheses in the definition of gap. Without those, gap would be a two-argument function, and this example would not work.

Similarly, property destructuring can also be used for function arguments:

```
julia> foo((; x, y)) = x + y
foo (generic function with 1 method)

julia> foo((x=1, y=2))
3

julia> struct A
           x
           y
       end

julia> foo(A(3, 4))
7
```

For anonymous functions, destructuring a single argument requires an extra comma:

```
julia> map(((x, y),) -> x + y, [(1, 2), (3, 4)])
2-element Vector{Int64}:
 3
 7
```

9.12 Varargs Functions

It is often convenient to be able to write functions taking an arbitrary number of arguments. Such functions are traditionally known as "varargs" functions, which is short for "variable number of arguments". You can define a varargs function by following the last positional argument with an ellipsis:

```
julia> bar(a, b, x...) = (a, b, x)
bar (generic function with 1 method)
```

The variables a and b are bound to the first two argument values as usual, and the variable x is bound to an iterable collection of the zero or more values passed to bar after its first two arguments:

```
julia> bar(1, 2)
(1, 2, ())

julia> bar(1, 2, 3)
(1, 2, (3,))

julia> bar(1, 2, 3, 4)
(1, 2, (3, 4))

julia> bar(1, 2, 3, 4, 5, 6)
(1, 2, (3, 4, 5, 6))
```

In all these cases, x is bound to a tuple of the trailing values passed to bar.

It is possible to constrain the number of values passed as a variable argument; this will be discussed later in Parametrically-constrained Varargs methods.

On the flip side, it is often handy to "splat" the values contained in an iterable collection into a function call as individual arguments. To do this, one also uses ... but in the function call instead:

```
julia> x = (3, 4)
(3, 4)

julia> bar(1, 2, x...)
(1, 2, (3, 4))
```

In this case a tuple of values is spliced into a varargs call precisely where the variable number of arguments go. This need not be the case, however:

```
julia> x = (2, 3, 4)
(2, 3, 4)

julia> bar(1, x...)
(1, 2, (3, 4))

julia> x = (1, 2, 3, 4)
(1, 2, 3, 4)

julia> bar(x...)
(1, 2, (3, 4))
```

Furthermore, the iterable object splatted into a function call need not be a tuple:

```
julia> x = [3, 4]
2-element Vector{Int64}:
 3
 4

julia> bar(1, 2, x...)
(1, 2, (3, 4))

julia> x = [1, 2, 3, 4]
4-element Vector{Int64}:
 1
 2
 3
 4

julia> bar(x...)
(1, 2, (3, 4))
```

Also, the function that arguments are splatted into need not be a varargs function (although it often is):

```
julia> baz(a, b) = a + b;

julia> args = [1, 2]
2-element Vector{Int64}:
 1
 2

julia> baz(args...)
3

julia> args = [1, 2, 3]
3-element Vector{Int64}:
 1
 2
 3

julia> baz(args...)
ERROR: MethodError: no method matching baz(::Int64, ::Int64, ::Int64)
The function `baz` exists, but no method is defined for this combination of argument types.

Closest candidates are:
  baz(::Any, ::Any)
   @ Main none:1

Stacktrace:
[...]
```

As you can see, if the wrong number of elements are in the splatted container, then the function call will fail, just as it would if too many arguments were given explicitly.

9.13 Optional Arguments

It is often possible to provide sensible default values for function arguments. This can save users from having to pass every argument on every call. For example, the function `Date(y, [m, d])` from `Dates` module constructs a `Date` type for a given year `y`, month `m` and day `d`. However, `m` and `d` arguments are optional and their default value is 1. This behavior can be expressed concisely as:

```
julia> using Dates

julia> function date(y::Int64, m::Int64=1, d::Int64=1)
           err = Dates.validargs(Date, y, m, d)
           err === nothing || throw(err)
           return Date(Dates.UTD(Dates.totaldays(y, m, d)))
       end
date (generic function with 3 methods)
```

Observe, that this definition calls another method of the `Date` function that takes one argument of type `UTInstant{Day}`.

With this definition, the function can be called with either one, two or three arguments, and 1 is automatically passed when only one or two of the arguments are specified:

```
julia> date(2000, 12, 12)
2000-12-12

julia> date(2000, 12)
2000-12-01

julia> date(2000)
2000-01-01
```

Optional arguments are actually just a convenient syntax for writing multiple method definitions with different numbers of arguments (see Note on Optional and keyword Arguments). This can be checked for our `date` function example by calling the `methods` function:

```
julia> methods(date)
# 3 methods for generic function "date" from Main:
 [1] date(y::Int64, m::Int64, d::Int64)
     @ REPL[2]:1
 [2] date(y::Int64, m::Int64)
     @ REPL[2]:1
 [3] date(y::Int64)
     @ REPL[2]:1
```

9.14 Keyword Arguments

Some functions need a large number of arguments, or have a large number of behaviors. Remembering how to call such functions can be difficult. Keyword arguments can make these complex interfaces easier to use and extend by allowing arguments to be identified by name instead of only by position.

For example, consider a function `plot` that plots a line. This function might have many options, for controlling line style, width, color, and so on. If it accepts keyword arguments, a possible call might look like `plot(x, y, width=2)`, where we have chosen to specify only line width. Notice that this serves two purposes. The call is easier to read, since we can label an argument with its meaning. It also becomes possible to pass any subset of a large number of arguments, in any order.

Functions with keyword arguments are defined using a semicolon in the signature:

```
function plot(x, y; style="solid", width=1, color="black")
    ###
end
```

When the function is called, the semicolon is optional: one can either call `plot(x, y, width=2)` or `plot(x, y; width=2)`, but the former style is more common. An explicit semicolon is required only for passing varargs or computed keywords as described below.

Keyword argument default values are evaluated only when necessary (when a corresponding keyword argument is not passed), and in left-to-right order. Therefore default expressions may refer to prior keyword arguments.

The types of keyword arguments can be made explicit as follows:

```
function f(; x::Int=1)
    ###
end
```

Keyword arguments can also be used in varargs functions:

```
function plot(x...; style="solid")
    ###
end
```

Extra keyword arguments can be collected using `...`, as in varargs functions:

```
function f(x; y=0, kwargs...)
    ###
end
```

Inside `f`, `kwargs` will be an immutable key-value iterator over a named tuple. Named tuples (as well as dictionaries with keys of `Symbol`, and other iterators yielding two-value collections with symbol as first values) can be passed as keyword arguments using a semicolon in a call, e.g. `f(x, z=1; kwargs...)`.

If a keyword argument is not assigned a default value in the method definition, then it is *required*: an `UndefKeywordError` exception will be thrown if the caller does not assign it a value:

```
function f(x; y)
    ###
end
f(3, y=5) # ok, y is assigned
f(3)      # throws UndefKeywordError(:y)
```

One can also pass `key => value` expressions after a semicolon. For example, `plot(x, y; :width => 2)` is equivalent to `plot(x, y, width=2)`. This is useful in situations where the keyword name is computed at runtime.

When a bare identifier or dot expression occurs after a semicolon, the keyword argument name is implied by the identifier or field name. For example `plot(x, y; width)` is equivalent to `plot(x, y; width=width)` and `plot(x, y; options.width)` is equivalent to `plot(x, y; width=options.width)`.

The nature of keyword arguments makes it possible to specify the same argument more than once. For example, in the call `plot(x, y; options..., width=2)` it is possible that the `options` structure also contains a value for `width`. In such a case the rightmost occurrence takes precedence; in this example, `width` is certain to have the value 2. However, explicitly specifying the same keyword argument multiple times, for example `plot(x, y, width=2, width=3)`, is not allowed and results in a syntax error.

9.15 Evaluation Scope of Default Values

When optional and keyword argument default expressions are evaluated, only *previous* arguments are in scope. For example, given this definition:

```
function f(x, a=b, b=1)
    ###
end
```

the b in a=b refers to a b in an outer scope, not the subsequent argument b.

9.16 Do-Block Syntax for Function Arguments

Passing functions as arguments to other functions is a powerful technique, but the syntax for it is not always convenient. Such calls are especially awkward to write when the function argument requires multiple lines. As an example, consider calling `map` on a function with several cases:

```
map(x->begin
           if x < 0 && iseven(x)
               return 0
           elseif x == 0
               return 1
           else
               return x
           end
       end,
    [A, B, C])
```

Julia provides a reserved word do for rewriting this code more clearly:

```
map([A, B, C]) do x
    if x < 0 && iseven(x)
        return 0
    elseif x == 0
        return 1
    else
```

```
        return x
    end
end
```

The `do x` syntax creates an anonymous function with argument `x` and passes the anonymous function as the first argument to the "outer" function - `map` in this example. Similarly, `do a,b` would create a two-argument anonymous function. Note that `do (a,b)` would create a one-argument anonymous function, whose argument is a tuple to be deconstructed. A plain `do` would declare that what follows is an anonymous function of the form `() -> ...`.

How these arguments are initialized depends on the "outer" function; here, `map` will sequentially set `x` to `A`, `B`, `C`, calling the anonymous function on each, just as would happen in the syntax `map(func, [A, B, C])`.

This syntax makes it easier to use functions to effectively extend the language, since calls look like normal code blocks. There are many possible uses quite different from `map`, such as managing system state. For example, there is a version of `open` that runs code ensuring that the opened file is eventually closed:

```
open("outfile", "w") do io
    write(io, data)
end
```

This is accomplished by the following definition:

```
function open(f::Function, args...)
    io = open(args...)
    try
        f(io)
    finally
        close(io)
    end
end
```

Here, `open` first opens the file for writing and then passes the resulting output stream to the anonymous function you defined in the `do ... end` block. After your function exits, `open` will make sure that the stream is properly closed, regardless of whether your function exited normally or threw an exception. (The `try/finally` construct will be described in Control Flow.)

With the `do` block syntax, it helps to check the documentation or implementation to know how the arguments of the user function are initialized.

A `do` block, like any other inner function, can "capture" variables from its enclosing scope. For example, the variable `data` in the above example of `open...do` is captured from the outer scope. Captured variables can create performance challenges as discussed in performance tips.

9.17 Function composition and piping

Functions in Julia can be combined by composing or piping (chaining) them together.

Function composition is when you combine functions together and apply the resulting composition to arguments. You use the function composition operator (`∘`) to compose the functions, so `(f ∘ g)(args...; kw...)` is the same as `f(g(args...; kw...))`.

You can type the composition operator at the REPL and suitably-configured editors using `\circ<tab>`.

For example, the `sqrt` and `+` functions can be composed like this:

```
julia> (sqrt ∘ +)(3, 6)
3.0
```

This adds the numbers first, then finds the square root of the result.

The next example composes three functions and maps the result over an array of strings:

```
julia> map(first ∘ reverse ∘ uppercase, split("you can compose functions like this"))
6-element Vector{Char}:
 'U': ASCII/Unicode U+0055 (category Lu: Letter, uppercase)
 'N': ASCII/Unicode U+004E (category Lu: Letter, uppercase)
 'E': ASCII/Unicode U+0045 (category Lu: Letter, uppercase)
 'S': ASCII/Unicode U+0053 (category Lu: Letter, uppercase)
 'E': ASCII/Unicode U+0045 (category Lu: Letter, uppercase)
 'S': ASCII/Unicode U+0053 (category Lu: Letter, uppercase)
```

Function chaining (sometimes called "piping" or "using a pipe" to send data to a subsequent function) is when you apply a function to the previous function's output:

```
julia> 1:10 |> sum |> sqrt
7.416198487095663
```

Here, the total produced by `sum` is passed to the `sqrt` function. The equivalent composition would be:

```
julia> (sqrt ∘ sum)(1:10)
7.416198487095663
```

The pipe operator can also be used with broadcasting, as `.|>`, to provide a useful combination of the chaining/piping and dot vectorization syntax (described below).

```
julia> ["a", "list", "of", "strings"] .|> [uppercase, reverse, titlecase, length]
4-element Vector{Any}:
  "A"
  "tsil"
  "Of"
 7
```

When combining pipes with anonymous functions, parentheses must be used if subsequent pipes are not to be parsed as part of the anonymous function's body. Compare:

```
julia> 1:3 .|> (x -> x^2) |> sum |> sqrt
3.7416573867739413

julia> 1:3 .|> x -> x^2 |> sum |> sqrt
3-element Vector{Float64}:
 1.0
 2.0
 3.0
```

9.18 Dot Syntax for Vectorizing Functions

In technical-computing languages, it is common to have "vectorized" versions of functions, which simply apply a given function `f(x)` to each element of an array `A` to yield a new array via `f(A)`. This kind of syntax is convenient for data processing, but in other languages vectorization is also often required for performance: if loops are slow, the "vectorized" version of a function can call fast library code written in a low-level language. In Julia, vectorized functions are *not* required for performance, and indeed it is often beneficial to write your own loops (see Performance Tips), but they can still be convenient. Therefore, *any* Julia function `f` can be applied elementwise to any array (or other collection) with the syntax `f.(A)`. For example, `sin` can be applied to all elements in the vector `A` like so:

```
julia> A = [1.0, 2.0, 3.0]
3-element Vector{Float64}:
 1.0
 2.0
 3.0

julia> sin.(A)
3-element Vector{Float64}:
 0.8414709848078965
 0.9092974268256817
 0.1411200080598672
```

Of course, you can omit the dot if you write a specialized "vector" method of `f`, e.g. via `f(A::AbstractArray) = map(f, A)`, and this is just as efficient as `f.(A)`. The advantage of the `f.(A)` syntax is that which functions are vectorizable need not be decided upon in advance by the library writer.

More generally, `f.(args...)` is actually equivalent to `broadcast(f, args...)`, which allows you to operate on multiple arrays (even of different shapes), or a mix of arrays and scalars (see Broadcasting). For example, if you have `f(x, y) = 3x + 4y`, then `f.(pi, A)` will return a new array consisting of `f(pi,a)` for each `a` in `A`, and `f.(vector1, vector2)` will return a new vector consisting of `f(vector1[i], vector2[i])` for each index `i` (throwing an exception if the vectors have different length).

```
julia> f(x, y) = 3x + 4y;

julia> A = [1.0, 2.0, 3.0];

julia> B = [4.0, 5.0, 6.0];

julia> f.(pi, A)
3-element Vector{Float64}:
 13.42477796076938
 17.42477796076938
 21.42477796076938

julia> f.(A, B)
3-element Vector{Float64}:
 19.0
 26.0
 33.0
```

Keyword arguments are not broadcasted over, but are simply passed through to each call of the function. For example, `round.(x, digits=3)` is equivalent to `broadcast(x -> round(x, digits=3), x)`.

Moreover, *nested* `f.(args...)` calls are *fused* into a single `broadcast` loop. For example, `sin.(cos.(X))` is equivalent to `broadcast(x -> sin(cos(x)), X)`, similar to `[sin(cos(x)) for x in X]`: there is only a single loop over `X`, and a single array is allocated for the result. [In contrast, `sin(cos(X))` in a typical "vectorized" language would first allocate one temporary array for `tmp=cos(X)`, and then compute `sin(tmp)` in a separate loop, allocating a second array.] This loop fusion is not a compiler optimization that may or may not occur, it is a *syntactic guarantee* whenever nested `f.(args...)` calls are encountered. Technically, the fusion stops as soon as a "non-dot" function call is encountered; for example, in `sin.(sort(cos.(X)))` the `sin` and `cos` loops cannot be merged because of the intervening `sort` function.

Finally, the maximum efficiency is typically achieved when the output array of a vectorized operation is *pre-allocated*, so that repeated calls do not allocate new arrays over and over again for the results (see Pre-allocate outputs). A convenient syntax for this is `X .= ...`, which is equivalent to `broadcast!(identity, X, ...)` except that, as above, the `broadcast!` loop is fused with any nested "dot" calls. For example, `X .= sin.(Y)` is equivalent to `broadcast!(sin, X, Y)`, overwriting `X` with `sin.(Y)` in-place. If the left-hand side is an array-indexing expression, e.g. `X[begin+1:end] .= sin.(Y)`, then it translates to `broadcast!` on a `view`, e.g. `broadcast!(sin, view(X, firstindex(X)+1:lastindex(X)), Y)`, so that the left-hand side is updated in-place.

Since adding dots to many operations and function calls in an expression can be tedious and lead to code that is difficult to read, the macro `@.` is provided to convert *every* function call, operation, and assignment in an expression into the "dotted" version.

```
julia> Y = [1.0, 2.0, 3.0, 4.0];

julia> X = similar(Y); # pre-allocate output array

julia> @. X = sin(cos(Y)) # equivalent to X .= sin.(cos.(Y))
4-element Vector{Float64}:
  0.5143952585235492
 -0.4042391538522658
 -0.8360218615377305
 -0.6080830096407656
```

Binary (or unary) operators like `.+` are handled with the same mechanism: they are equivalent to `broadcast` calls and are fused with other nested "dot" calls. `X .+= Y` etcetera is equivalent to `X .= X .+ Y` and results in a fused in-place assignment; see also dot operators.

You can also combine dot operations with function chaining using `|>`, as in this example:

```
julia> 1:5 .|> [x->x^2, inv, x->2*x, -, isodd]
5-element Vector{Real}:
    1
    0.5
    6
   -4
 true
```

All functions in the fused broadcast are always called for every element of the result. Thus `X .+ σ .* randn.()` will add a mask of independent and identically sampled random values to each element of the

array X, but X `.+ σ .*` `randn()` will add the *same* random sample to each element. In cases where the fused computation is constant along one or more axes of the broadcast iteration, it may be possible to leverage a space-time tradeoff and allocate intermediate values to reduce the number of computations. See more at performance tips.

9.19 Further Reading

We should mention here that this is far from a complete picture of defining functions. Julia has a sophisticated type system and allows multiple dispatch on argument types. None of the examples given here provide any type annotations on their arguments, meaning that they are applicable to all types of arguments. The type system is described in Types and defining a function in terms of methods chosen by multiple dispatch on run-time argument types is described in Methods.

Chapter 10

Control Flow

Julia provides a variety of control flow constructs:

- Compound Expressions: `begin` and `;`.
- Conditional Evaluation: `if-elseif-else` and `?:` (ternary operator).
- Short-Circuit Evaluation: logical operators && (“and”) and || (“or”), and also chained comparisons.
- Repeated Evaluation: Loops: `while` and `for`.
- Exception Handling: `try-catch`, `error` and `throw`.
- Tasks (aka Coroutines): `yieldto`.

The first five control flow mechanisms are standard to high-level programming languages. Tasks are not so standard: they provide non-local control flow, making it possible to switch between temporarily-suspended computations. This is a powerful construct: both exception handling and cooperative multitasking are implemented in Julia using tasks. Everyday programming requires no direct usage of tasks, but certain problems can be solved much more easily by using tasks.

10.1 Compound Expressions

Sometimes it is convenient to have a single expression which evaluates several subexpressions in order, returning the value of the last subexpression as its value. There are two Julia constructs that accomplish this: `begin` blocks and `;` chains. The value of both compound expression constructs is that of the last subexpression. Here's an example of a `begin` block:

```
julia> z = begin
           x = 1
           y = 2
           x + y
       end
3
```

Since these are fairly small, simple expressions, they could easily be placed onto a single line, which is where the `;` chain syntax comes in handy:

```
julia> z = (x = 1; y = 2; x + y)
3
```

This syntax is particularly useful with the terse single-line function definition form introduced in Functions. Although it is typical, there is no requirement that `begin` blocks be multiline or that `;` chains be single-line:

```
julia> begin x = 1; y = 2; x + y end
3

julia> (x = 1;
        y = 2;
        x + y)
3
```

10.2 Conditional Evaluation

Conditional evaluation allows portions of code to be evaluated or not evaluated depending on the value of a boolean expression. Here is the anatomy of the `if`-`elseif`-`else` conditional syntax:

```
if x < y
    println("x is less than y")
elseif x > y
    println("x is greater than y")
else
    println("x is equal to y")
end
```

If the condition expression `x < y` is `true`, then the corresponding block is evaluated; otherwise the condition expression `x > y` is evaluated, and if it is `true`, the corresponding block is evaluated; if neither expression is true, the `else` block is evaluated. Here it is in action:

```
julia> function test(x, y)
           if x < y
               println("x is less than y")
           elseif x > y
               println("x is greater than y")
           else
               println("x is equal to y")
           end
       end
test (generic function with 1 method)

julia> test(1, 2)
x is less than y

julia> test(2, 1)
x is greater than y

julia> test(1, 1)
x is equal to y
```

The `elseif` and `else` blocks are optional, and as many `elseif` blocks as desired can be used. The condition expressions in the `if`-`elseif`-`else` construct are evaluated until the first one evaluates to `true`, after which the associated block is evaluated, and no further condition expressions or blocks are evaluated.

`if` blocks are "leaky", i.e. they do not introduce a local scope. This means that new variables defined inside the `if` clauses can be used after the `if` block, even if they weren't defined before. So, we could have defined the `test` function above as

```
julia> function test(x,y)
           if x < y
               relation = "less than"
           elseif x == y
               relation = "equal to"
           else
               relation = "greater than"
           end
           println("x is ", relation, " y.")
       end
test (generic function with 1 method)

julia> test(2, 1)
x is greater than y.
```

The variable `relation` is declared inside the `if` block, but used outside. However, when depending on this behavior, make sure all possible code paths define a value for the variable. The following change to the above function results in a runtime error

```
julia> function test(x,y)
           if x < y
               relation = "less than"
           elseif x == y
               relation = "equal to"
           end
           println("x is ", relation, " y.")
       end
test (generic function with 1 method)

julia> test(1,2)
x is less than y.

julia> test(2,1)
ERROR: UndefVarError: `relation` not defined in local scope
Stacktrace:
 [1] test(::Int64, ::Int64) at ./none:7
```

`if` blocks also return a value, which may seem unintuitive to users coming from many other languages. This value is simply the return value of the last executed statement in the branch that was chosen, so

```
julia> x = 3
3
```

```
julia> if x > 0
           "positive!"
       else
           "negative..."
       end
"positive!"
```

Note that very short conditional statements (one-liners) are frequently expressed using Short-Circuit Evaluation in Julia, as outlined in the next section.

Unlike C, MATLAB, Perl, Python, and Ruby – but like Java, and a few other stricter, typed languages – it is an error if the value of a conditional expression is anything but `true` or `false`:

```
julia> if 1
           println("true")
       end
ERROR: TypeError: non-boolean (Int64) used in boolean context
```

This error indicates that the conditional was of the wrong type: `Int64` rather than the required `Bool`.

The so-called "ternary operator", `?:`, is closely related to the `if-elseif-else` syntax, but is used where a conditional choice between single expression values is required, as opposed to conditional execution of longer blocks of code. It gets its name from being the only operator in most languages taking three operands:

```
a ? b : c
```

The expression `a`, before the `?`, is a condition expression, and the ternary operation evaluates the expression `b`, before the `:`, if the condition `a` is `true` or the expression `c`, after the `:`, if it is `false`. Note that the spaces around `?` and `:` are mandatory: an expression like `a?b:c` is not a valid ternary expression (but a newline is acceptable after both the `?` and the `:`).

The easiest way to understand this behavior is to see an example. In the previous example, the `println` call is shared by all three branches: the only real choice is which literal string to print. This could be written more concisely using the ternary operator. For the sake of clarity, let's try a two-way version first:

```
julia> x = 1; y = 2;

julia> println(x < y ? "less than" : "not less than")
less than

julia> x = 1; y = 0;

julia> println(x < y ? "less than" : "not less than")
not less than
```

If the expression `x < y` is true, the entire ternary operator expression evaluates to the string `"less than"` and otherwise it evaluates to the string `"not less than"`. The original three-way example requires chaining multiple uses of the ternary operator together:

```
julia> test(x, y) = println(x < y ? "x is less than y"    :
                            x > y ? "x is greater than y" : "x is equal to y")
test (generic function with 1 method)

julia> test(1, 2)
x is less than y

julia> test(2, 1)
x is greater than y

julia> test(1, 1)
x is equal to y
```

To facilitate chaining, the operator associates from right to left.

It is significant that like `if-elseif-else`, the expressions before and after the `:` are only evaluated if the condition expression evaluates to `true` or `false`, respectively:

```
julia> v(x) = (println(x); x)
v (generic function with 1 method)

julia> 1 < 2 ? v("yes") : v("no")
yes
"yes"

julia> 1 > 2 ? v("yes") : v("no")
no
"no"
```

10.3 Short-Circuit Evaluation

The && and || operators in Julia correspond to logical "and" and "or" operations, respectively, and are typically used for this purpose. However, they have an additional property of *short-circuit* evaluation: they don't necessarily evaluate their second argument, as explained below. (There are also bitwise & and | operators that can be used as logical "and" and "or" *without* short-circuit behavior, but beware that & and | have higher precedence than && and || for evaluation order.)

Short-circuit evaluation is quite similar to conditional evaluation. The behavior is found in most imperative programming languages having the && and || boolean operators: in a series of boolean expressions connected by these operators, only the minimum number of expressions are evaluated as are necessary to determine the final boolean value of the entire chain. Some languages (like Python) refer to them as `and` (&&) and `or` (||). Explicitly, this means that:

- In the expression `a && b`, the subexpression `b` is only evaluated if `a` evaluates to `true`.
- In the expression `a || b`, the subexpression `b` is only evaluated if `a` evaluates to `false`.

The reasoning is that `a && b` must be `false` if `a` is `false`, regardless of the value of `b`, and likewise, the value of `a || b` must be true if `a` is `true`, regardless of the value of `b`. Both && and || associate to the right, but && has higher precedence than || does. It's easy to experiment with this behavior:

```
julia> t(x) = (println(x); true)
t (generic function with 1 method)

julia> f(x) = (println(x); false)
f (generic function with 1 method)

julia> t(1) && t(2)
1
2
true

julia> t(1) && f(2)
1
2
false

julia> f(1) && t(2)
1
false

julia> f(1) && f(2)
1
false

julia> t(1) || t(2)
1
true

julia> t(1) || f(2)
1
true

julia> f(1) || t(2)
1
2
true

julia> f(1) || f(2)
1
2
false
```

You can easily experiment in the same way with the associativity and precedence of various combinations of && and || operators.

This behavior is frequently used in Julia to form an alternative to very short `if` statements. Instead of `if` <cond> <statement> `end`, one can write <cond> && <statement> (which could be read as: <cond> *and then* <statement>). Similarly, instead of `if !` <cond> <statement> `end`, one can write <cond> || <statement> (which could be read as: <cond> *or else* <statement>).

For example, a recursive factorial routine could be defined like this:

```
julia> function fact(n::Int)
```

```
            n >= 0 || error("n must be non-negative")
            n == 0 && return 1
            n * fact(n-1)
        end
fact (generic function with 1 method)

julia> fact(5)
120

julia> fact(0)
1

julia> fact(-1)
ERROR: n must be non-negative
Stacktrace:
 [1] error at ./error.jl:33 [inlined]
 [2] fact(::Int64) at ./none:2
 [3] top-level scope
```

Boolean operations *without* short-circuit evaluation can be done with the bitwise boolean operators introduced in Mathematical Operations and Elementary Functions: `&` and `|`. These are normal functions, which happen to support infix operator syntax, but always evaluate their arguments:

```
julia> f(1) & t(2)
1
2
false

julia> t(1) | t(2)
1
2
true
```

Just like condition expressions used in `if`, `elseif` or the ternary operator, the operands of `&&` or `||` must be boolean values (`true` or `false`). Using a non-boolean value anywhere except for the last entry in a conditional chain is an error:

```
julia> 1 && true
ERROR: TypeError: non-boolean (Int64) used in boolean context
```

On the other hand, any type of expression can be used at the end of a conditional chain. It will be evaluated and returned depending on the preceding conditionals:

```
julia> true && (x = (1, 2, 3))
(1, 2, 3)

julia> false && (x = (1, 2, 3))
false
```

10.4 Repeated Evaluation: Loops

There are two constructs for repeated evaluation of expressions: the `while` loop and the `for` loop. Here is an example of a `while` loop:

```
julia> i = 1;

julia> while i <= 3
           println(i)
           global i += 1
       end
1
2
3
```

The `while` loop evaluates the condition expression (`i <= 3` in this case), and as long it remains `true`, keeps also evaluating the body of the `while` loop. If the condition expression is `false` when the `while` loop is first reached, the body is never evaluated.

The `for` loop makes common repeated evaluation idioms easier to write. Since counting up and down like the above `while` loop does is so common, it can be expressed more concisely with a `for` loop:

```
julia> for i = 1:3
           println(i)
       end
1
2
3
```

Here the `1:3` is a `range` object, representing the sequence of numbers 1, 2, 3. The `for` loop iterates through these values, assigning each one in turn to the variable `i`. In general, the `for` construct can loop over any "iterable" object (or "container"), from a range like `1:3` or `1:3:13` (a `StepRange` indicating every 3rd integer 1, 4, 7, ..., 13) to more generic containers like arrays, including iterators defined by user code or external packages. For containers other than ranges, the alternative (but fully equivalent) keyword `in` or `∈` is typically used instead of `=`, since it makes the code read more clearly:

```
julia> for i in [1,4,0]
           println(i)
       end
1
4
0

julia> for s ∈ ["foo","bar","baz"]
           println(s)
       end
foo
bar
baz
```

Various types of iterable containers will be introduced and discussed in later sections of the manual (see, e.g., Multi-dimensional Arrays).

One rather important distinction between the previous `while` loop form and the `for` loop form is the scope during which the variable is visible. A `for` loop always introduces a new iteration variable in its body, regardless of whether a variable of the same name exists in the enclosing scope. This implies that on the one hand `i` need not be declared before the loop. On the other hand it will not be visible outside the loop, nor will an outside variable of the same name be affected. You'll either need a new interactive session instance or a different variable name to test this:

```
julia> for j = 1:3
           println(j)
       end
1
2
3

julia> j
ERROR: UndefVarError: `j` not defined in `Main`
```

```
julia> j = 0;

julia> for j = 1:3
           println(j)
       end
1
2
3

julia> j
0
```

Use `for outer` to modify the latter behavior and reuse an existing local variable.

See Scope of Variables for a detailed explanation of variable scope, `outer`, and how it works in Julia.

It is sometimes convenient to terminate the repetition of a `while` before the test condition is falsified or stop iterating in a `for` loop before the end of the iterable object is reached. This can be accomplished with the `break` keyword:

```
julia> i = 1;

julia> while true
           println(i)
           if i >= 3
               break
           end
           global i += 1
       end
1
2
3
```

```
julia> for j = 1:1000
           println(j)
           if j >= 3
               break
           end
       end
1
2
3
```

Without the `break` keyword, the above `while` loop would never terminate on its own, and the `for` loop would iterate up to 1000. These loops are both exited early by using `break`.

In other circumstances, it is handy to be able to stop an iteration and move on to the next one immediately. The `continue` keyword accomplishes this:

```
julia> for i = 1:10
           if i % 3 != 0
               continue
           end
           println(i)
       end
3
6
9
```

This is a somewhat contrived example since we could produce the same behavior more clearly by negating the condition and placing the `println` call inside the `if` block. In realistic usage there is more code to be evaluated after the `continue`, and often there are multiple points from which one calls `continue`.

Multiple nested `for` loops can be combined into a single outer loop, forming the cartesian product of its iterables:

```
julia> for i = 1:2, j = 3:4
           println((i, j))
       end
(1, 3)
(1, 4)
(2, 3)
(2, 4)
```

With this syntax, iterables may still refer to outer loop variables; e.g. `for i = 1:n, j = 1:i` is valid. However a `break` statement inside such a loop exits the entire nest of loops, not just the inner one. Both variables (`i` and `j`) are set to their current iteration values each time the inner loop runs. Therefore, assignments to `i` will not be visible to subsequent iterations:

```
julia> for i = 1:2, j = 3:4
           println((i, j))
           i = 0
       end
```

```
(1, 3)
(1, 4)
(2, 3)
(2, 4)
```

If this example were rewritten to use a `for` keyword for each variable, then the output would be different: the second and fourth values would contain `0`.

Multiple containers can be iterated over at the same time in a single `for` loop using `zip`:

```
julia> for (j, k) in zip([1 2 3], [4 5 6 7])
           println((j,k))
       end
(1, 4)
(2, 5)
(3, 6)
```

Using `zip` will create an iterator that is a tuple containing the subiterators for the containers passed to it. The `zip` iterator will iterate over all subiterators in order, choosing the ith element of each subiterator in the ith iteration of the `for` loop. Once any of the subiterators run out, the `for` loop will stop.

10.5 Exception Handling

When an unexpected condition occurs, a function may be unable to return a reasonable value to its caller. In such cases, it may be best for the exceptional condition to either terminate the program while printing a diagnostic error message, or if the programmer has provided code to handle such exceptional circumstances then allow that code to take the appropriate action.

Built-in `Exceptions`

`Exceptions` are thrown when an unexpected condition has occurred. The built-in `Exceptions` listed below all interrupt the normal flow of control.

For example, the `sqrt` function throws a `DomainError` if applied to a negative real value:

```
julia> sqrt(-1)
ERROR: DomainError with -1.0:
sqrt was called with a negative real argument but will only return a complex result if called
↪  with a complex argument. Try sqrt(Complex(x)).
Stacktrace:
[...]
```

You may define your own exceptions in the following way:

```
julia> struct MyCustomException <: Exception end
```

Exception
ArgumentError
BoundsError
CompositeException
DimensionMismatch
DivideError
DomainError
EOFError
ErrorException
FieldError
InexactError
InitError
InterruptException
InvalidStateException
KeyError
LoadError
OutOfMemoryError
ReadOnlyMemoryError
RemoteException
MethodError
OverflowError
Meta.ParseError
SystemError
TypeError
UndefRefError
UndefVarError
StringIndexError

The `throw` function

Exceptions can be created explicitly with `throw`. For example, a function defined only for non-negative numbers could be written to `throw` a `DomainError` if the argument is negative:

```
julia> f(x) = x>=0 ? exp(-x) : throw(DomainError(x, "argument must be non-negative"))
f (generic function with 1 method)

julia> f(1)
0.36787944117144233

julia> f(-1)
ERROR: DomainError with -1:
argument must be non-negative
Stacktrace:
 [1] f(::Int64) at ./none:1
```

Note that `DomainError` without parentheses is not an exception, but a type of exception. It needs to be called to obtain an `Exception` object:

```
julia> typeof(DomainError(nothing)) <: Exception
```

```
true

julia> typeof(DomainError) <: Exception
false
```

Additionally, some exception types take one or more arguments that are used for error reporting:

```
julia> throw(UndefVarError(:x))
ERROR: UndefVarError: `x` not defined
```

This mechanism can be implemented easily by custom exception types following the way `UndefVarError` is written:

```
julia> struct MyUndefVarError <: Exception
           var::Symbol
       end

julia> Base.showerror(io::IO, e::MyUndefVarError) = print(io, e.var, " not defined")
```

Note

When writing an error message, it is preferred to make the first word lowercase. For example,

`size(A) == size(B) || throw(DimensionMismatch("size of A not equal to size of B"))`

is preferred over

`size(A) == size(B) || throw(DimensionMismatch("Size of A not equal to size of B")).`

However, sometimes it makes sense to keep the uppercase first letter, for instance if an argument to a function is a capital letter:

`size(A,1) == size(B,2) || throw(DimensionMismatch("A has first dimension...")).`

Errors

The `error` function is used to produce an `ErrorException` that interrupts the normal flow of control.

Suppose we want to stop execution immediately if the square root of a negative number is taken. To do this, we can define a fussy version of the `sqrt` function that raises an error if its argument is negative:

```
julia> fussy_sqrt(x) = x >= 0 ? sqrt(x) : error("negative x not allowed")
fussy_sqrt (generic function with 1 method)

julia> fussy_sqrt(2)
1.4142135623730951

julia> fussy_sqrt(-1)
ERROR: negative x not allowed
Stacktrace:
```

```
 [1] error at ./error.jl:33 [inlined]
 [2] fussy_sqrt(::Int64) at ./none:1
 [3] top-level scope
```

If `fussy_sqrt` is called with a negative value from another function, instead of trying to continue execution of the calling function, it returns immediately, displaying the error message in the interactive session:

```
julia> function verbose_fussy_sqrt(x)
           println("before fussy_sqrt")
           r = fussy_sqrt(x)
           println("after fussy_sqrt")
           return r
       end
verbose_fussy_sqrt (generic function with 1 method)

julia> verbose_fussy_sqrt(2)
before fussy_sqrt
after fussy_sqrt
1.4142135623730951

julia> verbose_fussy_sqrt(-1)
before fussy_sqrt
ERROR: negative x not allowed
Stacktrace:
 [1] error at ./error.jl:33 [inlined]
 [2] fussy_sqrt at ./none:1 [inlined]
 [3] verbose_fussy_sqrt(::Int64) at ./none:3
 [4] top-level scope
```

The `try/catch` statement

The `try/catch` statement allows for `Exceptions` to be tested for, and for the graceful handling of things that may ordinarily break your application. For example, in the below code the function for square root would normally throw an exception. By placing a `try/catch` block around it we can mitigate that here. You may choose how you wish to handle this exception, whether logging it, return a placeholder value or as in the case below where we just printed out a statement. One thing to think about when deciding how to handle unexpected situations is that using a `try/catch` block is much slower than using conditional branching to handle those situations. Below there are more examples of handling exceptions with a `try/catch` block:

```
julia> try
           sqrt("ten")
       catch e
           println("You should have entered a numeric value")
       end
You should have entered a numeric value
```

`try/catch` statements also allow the `Exception` to be saved in a variable. The following contrived example calculates the square root of the second element of x if x is indexable, otherwise assumes x is a real number and returns its square root:

```
julia> sqrt_second(x) = try
           sqrt(x[2])
       catch y
           if isa(y, DomainError)
               sqrt(complex(x[2], 0))
           elseif isa(y, BoundsError)
               sqrt(x)
           else
               rethrow() # ensure other exceptions can bubble up the call stack
           end
       end
sqrt_second (generic function with 1 method)

julia> sqrt_second([1 4])
2.0

julia> sqrt_second([1 -4])
0.0 + 2.0im

julia> sqrt_second(9)
3.0

julia> sqrt_second(-9)
ERROR: DomainError with -9.0:
sqrt was called with a negative real argument but will only return a complex result if called
↪  with a complex argument. Try sqrt(Complex(x)).
Stacktrace:
[...]

julia> sqrt_second([1 nothing])
ERROR: MethodError: no method matching sqrt(::Nothing)
The function `sqrt` exists, but no method is defined for this combination of argument types.
[...]
```

Use `rethrow` as above to continue unwinding the stack with the original exception so that higher-level exception handlers can deal with the exception. When filtering by exception type as above, it is often important to include `else rethrow()` so that other types of exceptions are not hidden from the caller.

Note that the symbol following `catch` will always be interpreted as a name for the exception, so care is needed when writing `try/catch` expressions on a single line. The following code will *not* work to return the value of x in case of an error:

```
try bad() catch x end
```

Instead, use a semicolon or insert a line break after `catch`:

```
try bad() catch; x end

try bad()
catch
    x
end
```

The power of the `try/catch` construct lies in the ability to unwind a deeply nested computation immediately to a much higher level in the stack of calling functions. There are situations where no error has occurred, but the ability to unwind the stack and pass a value to a higher level is desirable. Julia provides the `backtrace`, `catch_backtrace` and `current_exceptions` functions for more advanced error handling.

`else` Clauses

Julia 1.8

This functionality requires at least Julia 1.8.

In some cases, one may not only want to appropriately handle the error case, but also want to run some code only if the `try` block succeeds. For this, an `else` clause can be specified after the `catch` block that is run whenever no error was thrown previously. The advantage over including this code in the `try` block instead is that any further errors don't get silently caught by the `catch` clause.

```
local x
try
    x = read("file", String)
catch
    # handle read errors
else
    # do something with x
end
```

Note

The `try`, `catch`, `else`, and `finally` clauses each introduce their own scope blocks, so if a variable is only defined in the `try` block, it can not be accessed by the `else` or `finally` clause:

```
julia> try
           foo = 1
       catch
       else
           foo
       end
ERROR: UndefVarError: `foo` not defined in `Main`
Suggestion: check for spelling errors or missing imports.
```

Use the `local` keyword outside the `try` block to make the variable accessible from anywhere within the outer scope.

`finally` Clauses

In code that performs state changes or uses resources like files, there is typically clean-up work (such as closing files) that needs to be done when the code is finished. Exceptions potentially complicate this task, since they can cause a block of code to exit before reaching its normal end. The `finally` keyword provides a way to run some code when a given block of code exits, regardless of how it exits.

For example, here is how we can guarantee that an opened file is closed:

```
f = open("file")
try
    # operate on file f
finally
    close(f)
end
```

When control leaves the `try` block (for example due to a `return`, or just finishing normally), `close(f)` will be executed. If the `try` block exits due to an exception, the exception will continue propagating. A `catch` block may be combined with `try` and `finally` as well. In this case the `finally` block will run after `catch` has handled the error.

When evaluating a `try/catch/else/finally` expression, the value of the entire expression is the value of the last block executed, excluding the `finally` block. For example:

```
julia> try
           1
       finally
           2
       end
1

julia> try
           error("")
       catch
           1
       else
           2
       finally
           3
       end
1

julia> try
           0
       catch
           1
       else
           2
       finally
           3
       end
2
```

10.6 Tasks (aka Coroutines)

Tasks are a control flow feature that allows computations to be suspended and resumed in a flexible manner. We mention them here only for completeness; for a full discussion see Asynchronous Programming.

Chapter 11

Scope of Variables

The *scope* of a variable is the region of code within which a variable is accessible. Variable scoping helps avoid variable naming conflicts. The concept is intuitive: two functions can both have arguments called x without the two x's referring to the same thing. Similarly, there are many other cases where different blocks of code can use the same name without referring to the same thing. The rules for when the same variable name does or doesn't refer to the same thing are called scope rules; this section spells them out in detail.

Certain constructs in the language introduce *scope blocks*, which are regions of code that are eligible to be the scope of some set of variables. The scope of a variable cannot be an arbitrary set of source lines; instead, it will always line up with one of these blocks. There are two main types of scopes in Julia, *global scope* and *local scope*. The latter can be nested. There is also a distinction in Julia between constructs which introduce a "hard scope" and those which only introduce a "soft scope", which affects whether shadowing a global variable by the same name is allowed or not.

Summary

Variables defined in global scope may be undefined in inner local scopes, depending on where the code is run, in order to balance safety and convenience. The hard and soft local scoping rules define the interplay between global and local variables.

However, variables defined only in local scope behave consistently in all contexts. If the variable is already defined, it will be reused. If the variable is not defined, it will be made available to the current and inner scopes (but not outer scopes).

A Common Confusion

If you run into an unexpectedly undefined variable,

```
# Print the numbers 1 through 5
i = 0
while i < 5
    i += 1     # ERROR: UndefVarError: `i` not defined
    println(i)
end
```

a simple fix is to change all global variable definitions into local definitions by wrapping the code in a `let` block or `function`.

```
# Print the numbers 1 through 5
let i = 0
    while i < 5
        i += 1     # Now outer `i` is defined in the inner scope of the while loop
        println(i)
    end
end
```

This is a common source of confusion when writing procedural scripts, but it becomes a non-issue if code is moved inside functions or executed interactively in the REPL.

See also the `global` and `local` keywords to explicitly achieve any desired scoping behavior.

Scope Constructs

The constructs introducing scope blocks are:

Construct	Scope Type Introduced	Scope Types Able to Contain Construct
`module`, `baremodule`	global	global
`struct`	local (hard)	global
`macro`	local (hard)	global
`for`, `while`, `try`	local (soft)	global, local
`function`, `do`, `let`, comprehensions, generators	local (hard)	global, local

Notably missing from this table are begin blocks and if blocks which do *not* introduce new scopes. The three types of scopes follow somewhat different rules which will be explained below.

Julia uses lexical scoping, meaning that a function's scope does not inherit from its caller's scope, but from the scope in which the function was defined. For example, in the following code the `x` inside `foo` refers to the `x` in the global scope of its module `Bar`:

```
julia> module Bar
           x = 1
```

```
            foo() = x
        end;
```

and not a x in the scope where `foo` is used:

```
julia> import .Bar

julia> x = -1;

julia> Bar.foo()
1
```

Thus *lexical scope* means that what a variable in a particular piece of code refers to can be deduced from the code in which it appears alone and does not depend on how the program executes. A scope nested inside another scope can "see" variables in all the outer scopes in which it is contained. Outer scopes, on the other hand, cannot see variables in inner scopes.

11.1 Global Scope

Each module introduces a new global scope, separate from the global scope of all other modules—there is no all-encompassing global scope. Modules can introduce variables of other modules into their scope through the using or import statements or through qualified access using the dot-notation, i.e. each module is a so-called *namespace* as well as a first-class data structure associating names with values.

If a top-level expression contains a variable declaration with keyword `local`, then that variable is not accessible outside that expression. The variable inside the expression does not affect global variables of the same name. An example is to declare `local x` in a `begin` or `if` block at the top-level:

```
julia> x = 1
       begin
           local x = 0
           @show x
       end
       @show x;
x = 0
x = 1
```

Note that the interactive prompt (aka REPL) is in the global scope of the module `Main`.

11.2 Local Scope

A new local scope is introduced by most code blocks (see above table for a complete list). If such a block is syntactically nested inside of another local scope, the scope it creates is nested inside of all the local scopes that it appears within, which are all ultimately nested inside of the global scope of the module in which the code is evaluated. Variables in outer scopes are visible from any scope they contain — meaning that they can be read and written in inner scopes — unless there is a variable with the same name that "shadows" the outer variable of the same name. This is true even if the outer local is declared after (in the sense of textually below) an inner block. When we say that a variable "exists" in a given scope, this means that a variable by that name exists in any of the scopes that the current scope is nested inside of,

including the current one. If a variable's value is used in a local scope, but nothing with its name exists in this scope, it is assumed to be a global.

Some programming languages require explicitly declaring new variables before using them. Explicit declaration works in Julia too: in any local scope, writing `local x` declares a new local variable in that scope, regardless of whether there is already a variable named x in an outer scope or not. Declaring each new variable like this is somewhat verbose and tedious, however, so Julia, like many other languages, considers assignment to a variable name that doesn't already exist to implicitly declare that variable. If the current scope is global, the new variable is global; if the current scope is local, the new variable is local to the innermost local scope and will be visible inside of that scope but not outside of it. If you assign to an existing local, it *always* updates that existing local: you can only shadow a local by explicitly declaring a new local in a nested scope with the `local` keyword. In particular, this applies to variables assigned in inner functions, which may surprise users coming from Python where assignment in an inner function creates a new local unless the variable is explicitly declared to be non-local.

Mostly this is pretty intuitive, but as with many things that behave intuitively, the details are more subtle than one might naïvely imagine.

When `x = <value>` occurs in a local scope, Julia applies the following rules to decide what the expression means based on where the assignment expression occurs and what x already refers to at that location:

1. **Existing local:** If x is *already a local variable*, then the existing local x is assigned;
2. **Hard scope:** If x is *not already a local variable* and assignment occurs inside of any hard scope construct (i.e. within a `let` block, function, struct or macro body, comprehension, or generator), a new local named x is created in the scope of the assignment;
3. **Soft scope:** If x is *not already a local variable* and all of the scope constructs containing the assignment are soft scopes (loops, `try`/`catch` blocks), the behavior depends on whether the global variable x is defined:
   - if global x is *undefined*, a new local named x is created in the scope of the assignment;
   - if global x is *defined*, the assignment is considered ambiguous:
     - in *non-interactive* contexts (files, eval), an ambiguity warning is printed and a new local is created;
     - in *interactive* contexts (REPL, notebooks), the global variable x is assigned.

You may note that in non-interactive contexts the hard and soft scope behaviors are identical except that a warning is printed when an implicitly local variable (i.e. not declared with `local x`) shadows a global. In interactive contexts, the rules follow a more complex heuristic for the sake of convenience. This is covered in depth in examples that follow.

Now that you know the rules, let's look at some examples. Each example is assumed to be evaluated in a fresh REPL session so that the only globals in each snippet are the ones that are assigned in that block of code.

We'll begin with a nice and clear-cut situation—assignment inside of a hard scope, in this case a function body, when no local variable by that name already exists:

```
julia> function greet()
           x = "hello" # new local
```

```
            println(x)
        end
greet (generic function with 1 method)

julia> greet()
hello

julia> x # global
ERROR: UndefVarError: `x` not defined in `Main`
```

Inside of the `greet` function, the assignment `x = "hello"` causes `x` to be a new local variable in the function's scope. There are two relevant facts: the assignment occurs in local scope and there is no existing local `x` variable. Since `x` is local, it doesn't matter if there is a global named `x` or not. Here for example we define `x = 123` before defining and calling `greet`:

```
julia> x = 123 # global
123

julia> function greet()
            x = "hello" # new local
            println(x)
        end
greet (generic function with 1 method)

julia> greet()
hello

julia> x # global
123
```

Since the `x` in `greet` is local, the value (or lack thereof) of the global `x` is unaffected by calling `greet`. The hard scope rule doesn't care whether a global named `x` exists or not: assignment to `x` in a hard scope is local (unless `x` is declared global).

The next clear cut situation we'll consider is when there is already a local variable named `x`, in which case `x = <value>` always assigns to this existing local `x`. This is true whether the assignment occurs in the same local scope, an inner local scope in the same function body, or in the body of a function nested inside of another function, also known as a closure.

We'll use the `sum_to` function, which computes the sum of integers from one up to `n`, as an example:

```
function sum_to(n)
    s = 0 # new local
    for i = 1:n
        s = s + i # assign existing local
    end
    return s # same local
end
```

As in the previous example, the first assignment to `s` at the top of `sum_to` causes `s` to be a new local variable in the body of the function. The `for` loop has its own inner local scope within the function scope.

At the point where `s = s + i` occurs, `s` is already a local variable, so the assignment updates the existing `s` instead of creating a new local. We can test this out by calling `sum_to` in the REPL:

```
julia> function sum_to(n)
           s = 0 # new local
           for i = 1:n
               s = s + i # assign existing local
           end
           return s # same local
       end
sum_to (generic function with 1 method)

julia> sum_to(10)
55

julia> s # global
ERROR: UndefVarError: `s` not defined in `Main`
```

Since `s` is local to the function `sum_to`, calling the function has no effect on the global variable `s`. We can also see that the update `s = s + i` in the `for` loop must have updated the same `s` created by the initialization `s = 0` since we get the correct sum of 55 for the integers 1 through 10.

Let's dig into the fact that the `for` loop body has its own scope for a second by writing a slightly more verbose variation which we'll call `sum_to_def`, in which we save the sum `s + i` in a variable `t` before updating `s`:

```
julia> function sum_to_def(n)
           s = 0 # new local
           for i = 1:n
               t = s + i # new local `t`
               s = t # assign existing local `s`
           end
           return s, @isdefined(t)
       end
sum_to_def (generic function with 1 method)

julia> sum_to_def(10)
(55, false)
```

This version returns `s` as before but it also uses the `@isdefined` macro to return a boolean indicating whether there is a local variable named `t` defined in the function's outermost local scope. As you can see, there is no `t` defined outside of the `for` loop body. This is because of the hard scope rule again: since the assignment to `t` occurs inside of a function, which introduces a hard scope, the assignment causes `t` to become a new local variable in the local scope where it appears, i.e. inside of the loop body. Even if there were a global named `t`, it would make no difference—the hard scope rule isn't affected by anything in global scope.

Note that the local scope of a for loop body is no different from the local scope of an inner function. This means that we could rewrite this example so that the loop body is implemented as a call to an inner helper function and it behaves the same way:

```
julia> function sum_to_def_closure(n)
           function loop_body(i)
               t = s + i # new local `t`
               s = t # assign same local `s` as below
           end
           s = 0 # new local
           for i = 1:n
               loop_body(i)
           end
           return s, @isdefined(t)
       end
sum_to_def_closure (generic function with 1 method)

julia> sum_to_def_closure(10)
(55, false)
```

This example illustrates a couple of key points:

1. Inner function scopes are just like any other nested local scope. In particular, if a variable is already a local outside of an inner function and you assign to it in the inner function, the outer local variable is updated.
2. It doesn't matter if the definition of an outer local happens below where it is updated, the rule remains the same. The entire enclosing local scope is parsed and its locals determined before inner local meanings are resolved.

This design means that you can generally move code in or out of an inner function without changing its meaning, which facilitates a number of common idioms in the language using closures (see do blocks).

Let's move onto some more ambiguous cases covered by the soft scope rule. We'll explore this by extracting the bodies of the `greet` and `sum_to_def` functions into soft scope contexts. First, let's put the body of `greet` in a `for` loop—which is soft, rather than hard—and evaluate it in the REPL:

```
julia> for i = 1:3
           x = "hello" # new local
           println(x)
       end
hello
hello
hello

julia> x
ERROR: UndefVarError: `x` not defined in `Main`
```

Since the global `x` is not defined when the `for` loop is evaluated, the first clause of the soft scope rule applies and `x` is created as local to the `for` loop and therefore global `x` remains undefined after the loop executes. Next, let's consider the body of `sum_to_def` extracted into global scope, fixing its argument to `n = 10`

```
s = 0
```

```
for i = 1:10
    t = s + i
    s = t
end
s
@isdefined(t)
```

What does this code do? Hint: it's a trick question. The answer is "it depends." If this code is entered interactively, it behaves the same way it does in a function body. But if the code appears in a file, it prints an ambiguity warning and throws an undefined variable error. Let's see it working in the REPL first:

```
julia> s = 0 # global
0

julia> for i = 1:10
           t = s + i # new local `t`
           s = t # assign global `s`
       end

julia> s # global
55

julia> @isdefined(t) # global
false
```

The REPL approximates being in the body of a function by deciding whether assignment inside the loop assigns to a global or creates new local based on whether a global variable by that name is defined or not. If a global by the name exists, then the assignment updates it. If no global exists, then the assignment creates a new local variable. In this example we see both cases in action:

- There is no global named `t`, so `t = s + i` creates a new `t` that is local to the `for` loop;
- There is a global named `s`, so `s = t` assigns to it.

The second fact is why execution of the loop changes the global value of `s` and the first fact is why `t` is still undefined after the loop executes. Now, let's try evaluating this same code as though it were in a file instead:

```
julia> code = """
       s = 0 # global
       for i = 1:10
           t = s + i # new local `t`
           s = t # new local `s` with warning
       end
       s, # global
       @isdefined(t) # global
       """;

julia> include_string(Main, code)
```

```
┌ Warning: Assignment to `s` in soft scope is ambiguous because a global variable by the same
↪  name exists: `s` will be treated as a new local. Disambiguate by using `local s` to suppress
↪  this warning or `global s` to assign to the existing global variable.
└ @ string:4
ERROR: LoadError: UndefVarError: `s` not defined in local scope
```

Here we use `include_string`, to evaluate code as though it were the contents of a file. We could also save code to a file and then call `include` on that file—the result would be the same. As you can see, this behaves quite different from evaluating the same code in the REPL. Let's break down what's happening here:

- global `s` is defined with the value `0` before the loop is evaluated
- the assignment `s = t` occurs in a soft scope—a `for` loop outside of any function body or other hard scope construct
- therefore the second clause of the soft scope rule applies, and the assignment is ambiguous so a warning is emitted
- execution continues, making `s` local to the `for` loop body
- since `s` is local to the `for` loop, it is undefined when `t = s + i` is evaluated, causing an error
- evaluation stops there, but if it got to `s` and `@isdefined(t)`, it would return `0` and `false`.

This demonstrates some important aspects of scope: in a scope, each variable can only have one meaning, and that meaning is determined regardless of the order of expressions. The presence of the expression `s = t` in the loop causes `s` to be local to the loop, which means that it is also local when it appears on the right hand side of `t = s + i`, even though that expression appears first and is evaluated first. One might imagine that the `s` on the first line of the loop could be global while the `s` on the second line of the loop is local, but that's not possible since the two lines are in the same scope block and each variable can only mean one thing in a given scope.

On Soft Scope

We have now covered all the local scope rules, but before wrapping up this section, perhaps a few words should be said about why the ambiguous soft scope case is handled differently in interactive and non-interactive contexts. There are two obvious questions one could ask:

1. Why doesn't it just work like the REPL everywhere?
2. Why doesn't it just work like in files everywhere? And maybe skip the warning?

In Julia ≤ 0.6, all global scopes did work like the current REPL: when `x = <value>` occurred in a loop (or `try`/`catch`, or `struct` body) but outside of a function body (or `let` block or comprehension), it was decided based on whether a global named `x` was defined or not whether `x` should be local to the loop. This behavior has the advantage of being intuitive and convenient since it approximates the behavior inside of a function body as closely as possible. In particular, it makes it easy to move code back and forth between a function body and the REPL when trying to debug the behavior of a function. However, it has some downsides. First, it's quite a complex behavior: many people over the years were confused about this behavior and

complained that it was complicated and hard both to explain and understand. Fair point. Second, and arguably worse, is that it's bad for programming "at scale." When you see a small piece of code in one place like this, it's quite clear what's going on:

```
s = 0
for i = 1:10
    s += i
end
```

Obviously the intention is to modify the existing global variable `s`. What else could it mean? However, not all real world code is so short or so clear. We found that code like the following often occurs in the wild:

```
x = 123

# much later
# maybe in a different file

for i = 1:10
    x = "hello"
    println(x)
end

# much later
# maybe in yet another file
# or maybe back in the first one where `x = 123`

y = x + 234
```

It's far less clear what should happen here. Since `x + "hello"` is a method error, it seems probable that the intention is for `x` to be local to the `for` loop. But runtime values and what methods happen to exist cannot be used to determine the scopes of variables. With the Julia ≤ 0.6 behavior, it's especially concerning that someone might have written the `for` loop first, had it working just fine, but later when someone else adds a new global far away—possibly in a different file—the code suddenly changes meaning and either breaks noisily or, worse still, silently does the wrong thing. This kind of "spooky action at a distance" is something that good programming language designs should prevent.

So in Julia 1.0, we simplified the rules for scope: in any local scope, assignment to a name that wasn't already a local variable created a new local variable. This eliminated the notion of soft scope entirely as well as removing the potential for spooky action. We uncovered and fixed a significant number of bugs due to the removal of soft scope, vindicating the choice to get rid of it. And there was much rejoicing! Well, no, not really. Because some people were angry that they now had to write:

```
s = 0
for i = 1:10
    global s += i
end
```

Do you see that `global` annotation in there? Hideous. Obviously this situation could not be tolerated. But seriously, there are two main issues with requiring `global` for this kind of top-level code:

1. It's no longer convenient to copy and paste the code from inside a function body into the REPL to debug it—you have to add `global` annotations and then remove them again to go back;
2. Beginners will write this kind of code without the `global` and have no idea why their code doesn't work—the error that they get is that `s` is undefined, which does not seem to enlighten anyone who happens to make this mistake.

As of Julia 1.5, this code works without the `global` annotation in interactive contexts like the REPL or Jupyter notebooks (just like Julia 0.6) and in files and other non-interactive contexts, it prints this very direct warning:

> Assignment to `s` in soft scope is ambiguous because a global variable by the same name exists: `s` will be treated as a new local. Disambiguate by using `local s` to suppress this warning or `global s` to assign to the existing global variable.

This addresses both issues while preserving the "programming at scale" benefits of the 1.0 behavior: global variables have no spooky effect on the meaning of code that may be far away; in the REPL copy-and-paste debugging works and beginners don't have any issues; any time someone either forgets a `global` annotation or accidentally shadows an existing global with a local in a soft scope, which would be confusing anyway, they get a nice clear warning.

An important property of this design is that any code that executes in a file without a warning will behave the same way in a fresh REPL. And on the flip side, if you take a REPL session and save it to file, if it behaves differently than it did in the REPL, then you will get a warning.

Let Blocks

`let` statements create a new *hard scope* block (see above) and introduce new variable bindings each time they run. The variable need not be immediately assigned:

```
julia> var1 = let x
           for i in 1:5
               (i == 4) && (x = i; break)
           end
           x
       end
4
```

Whereas assignments might reassign a new value to an existing value location, `let` always creates a new location. This difference is usually not important, and is only detectable in the case of variables that outlive their scope via closures. The `let` syntax accepts a comma-separated series of assignments and variable names:

```
julia> x, y, z = -1, -1, -1;

julia> let x = 1, z
           println("x: $x, y: $y") # x is local variable, y the global
           println("z: $z") # errors as z has not been assigned yet but is local
       end
x: 1, y: -1
ERROR: UndefVarError: `z` not defined in local scope
```

The assignments are evaluated in order, with each right-hand side evaluated in the scope before the new variable on the left-hand side has been introduced. Therefore it makes sense to write something like `let x = x` since the two `x` variables are distinct and have separate storage. Here is an example where the behavior of `let` is needed:

```
julia> Fs = Vector{Any}(undef, 2); i = 1;

julia> while i <= 2
           Fs[i] = ()->i
           global i += 1
       end

julia> Fs[1]()
3

julia> Fs[2]()
3
```

Here we create and store two closures that return variable `i`. However, it is always the same variable `i`, so the two closures behave identically. We can use `let` to create a new binding for `i`:

```
julia> Fs = Vector{Any}(undef, 2); i = 1;

julia> while i <= 2
           let i = i
               Fs[i] = ()->i
           end
           global i += 1
       end

julia> Fs[1]()
1

julia> Fs[2]()
2
```

Since the `begin` construct does not introduce a new scope, it can be useful to use a zero-argument `let` to just introduce a new scope block without creating any new bindings immediately:

```
julia> let
           local x = 1
           let
               local x = 2
           end
           x
       end
1
```

Since `let` introduces a new scope block, the inner local `x` is a different variable than the outer local `x`. This particular example is equivalent to:

```
julia> let x = 1
           let x = 2
           end
           x
       end
1
```

Loops and Comprehensions

In loops and comprehensions, new variables introduced in their body scopes are freshly allocated for each loop iteration, as if the loop body were surrounded by a `let` block, as demonstrated by this example:

```
julia> Fs = Vector{Any}(undef, 2);

julia> for j = 1:2
           Fs[j] = ()->j
       end

julia> Fs[1]()
1

julia> Fs[2]()
2
```

A `for` loop or comprehension iteration variable is always a new variable:

```
julia> function f()
           i = 0
           for i = 1:3
               # empty
           end
           return i
       end;

julia> f()
0
```

However, it is occasionally useful to reuse an existing local variable as the iteration variable. This can be done conveniently by adding the keyword `outer`:

```
julia> function f()
           i = 0
           for outer i = 1:3
               # empty
           end
           return i
       end;

julia> f()
3
```

11.3 Constants

A common use of variables is giving names to specific, unchanging values. Such variables are only assigned once. This intent can be conveyed to the compiler using the `const` keyword:

```
julia> const e  = 2.71828182845904523536;

julia> const pi = 3.14159265358979323846;
```

Multiple variables can be declared in a single `const` statement:

```
julia> const a, b = 1, 2
(1, 2)
```

The `const` declaration should only be used in global scope on globals. It is difficult for the compiler to optimize code involving global variables, since their values (or even their types) might change at almost any time. If a global variable will not change, adding a `const` declaration solves this performance problem.

Local constants are quite different. The compiler is able to determine automatically when a local variable is constant, so local constant declarations are not necessary, and in fact are currently not supported.

Special top-level assignments, such as those performed by the `function` and `struct` keywords, are constant by default.

Note that `const` only affects the variable binding; the variable may be bound to a mutable object (such as an array), and that object may still be modified. Additionally when one tries to assign a value to a variable that is declared constant the following scenarios are possible:

- Attempting to replace a constant without the const keyword is disallowed:

  ```
  julia> const x = 1.0
  1.0

  julia> x = 1
  ERROR: invalid assignment to constant x. This redefinition may be permitted using the
  ↪  `const` keyword.
  ```

- All other definitions of constants are permitted, but may cause significant re-compilation:

  ```
  julia> const y = 1.0
  1.0

  julia> const y = 2.0
  2.0
  ```

Julia 1.12

Prior to julia 1.12, redefinition of constants was poorly supported. It was restricted to redefinition of constants of the same type and could lead to observably incorrect behavior or crashes. Constant redefinition is highly discouraged in versions of julia prior to 1.12. See the manual for prior julia versions for further information.

11.4 Typed Globals

Julia 1.8

Support for typed globals was added in Julia 1.8

Similar to being declared as constants, global bindings can also be declared to always be of a constant type. This can either be done without assigning an actual value using the syntax `global x::T` or upon assignment as `x::T = 123`.

```
julia> x::Float64 = 2.718
2.718

julia> f() = x
f (generic function with 1 method)

julia> Base.return_types(f)
1-element Vector{Any}:
 Float64
```

For any assignment to a global, Julia will first try to convert it to the appropriate type using `convert`:

```
julia> global y::Int

julia> y = 1.0
1.0

julia> y
1

julia> y = 3.14
ERROR: InexactError: Int64(3.14)
Stacktrace:
[...]
```

The type does not need to be concrete, but annotations with abstract types typically have little performance benefit.

Once a global has either been assigned to or its type has been set, the binding type is not allowed to change:

```
julia> x = 1
1

julia> global x::Int
ERROR: cannot set type for global x. It already has a value or is already set to a different
↪  type.
Stacktrace:
[...]
```

Chapter 12

Types

Type systems have traditionally fallen into two quite different camps: static type systems, where every program expression must have a type computable before the execution of the program, and dynamic type systems, where nothing is known about types until run time, when the actual values manipulated by the program are available. Object orientation allows some flexibility in statically typed languages by letting code be written without the precise types of values being known at compile time. The ability to write code that can operate on different types is called polymorphism. All code in classic dynamically typed languages is polymorphic: only by explicitly checking types, or when objects fail to support operations at run-time, are the types of any values ever restricted.

Julia's type system is dynamic, but gains some of the advantages of static type systems by making it possible to indicate that certain values are of specific types. This can be of great assistance in generating efficient code, but even more significantly, it allows method dispatch on the types of function arguments to be deeply integrated with the language. Method dispatch is explored in detail in Methods, but is rooted in the type system presented here.

The default behavior in Julia when types are omitted is to allow values to be of any type. Thus, one can write many useful Julia functions without ever explicitly using types. When additional expressiveness is needed, however, it is easy to gradually introduce explicit type annotations into previously "untyped" code. Adding annotations serves three primary purposes: to take advantage of Julia's powerful multiple-dispatch mechanism, to improve human readability, and to catch programmer errors.

Describing Julia in the lingo of type systems, it is: dynamic, nominative and parametric. Generic types can be parameterized, and the hierarchical relationships between types are explicitly declared, rather than implied by compatible structure. One particularly distinctive feature of Julia's type system is that concrete types may not subtype each other: all concrete types are final and may only have abstract types as their supertypes. While this might at first seem unduly restrictive, it has many beneficial consequences with surprisingly few drawbacks. It turns out that being able to inherit behavior is much more important than being able to inherit structure, and inheriting both causes significant difficulties in traditional object-oriented languages. While concrete types do have abstract subtypes, there are only two examples of this (`Union{}` and `Type{T}`) and additional subtypes of concrete types cannot be declared.

Other high-level aspects of Julia's type system that should be mentioned up front are:

- There is no division between object and non-object values: all values in Julia are true objects having a type that belongs to a single, fully connected type graph, all nodes of which are equally first-class as types.

- There is no meaningful concept of a "compile-time type": the only type a value has is its actual type when the program is running. This is called a "run-time type" in object-oriented languages where the combination of static compilation with polymorphism makes this distinction significant.
- Only values, not variables, have types – variables are simply names bound to values, although for simplicity we may say "type of a variable" as shorthand for "type of the value to which a variable refers".
- Both abstract and concrete types can be parameterized by other types. They can also be parameterized by symbols, by values of any type for which `isbits` returns true (essentially, things like numbers and bools that are stored like C types or `structs` with no pointers to other objects), and also by tuples thereof. Type parameters may be omitted when they do not need to be referenced or restricted.

Julia's type system is designed to be powerful and expressive, yet clear, intuitive and unobtrusive. Many Julia programmers may never feel the need to write code that explicitly uses types. Some kinds of programming, however, become clearer, simpler, faster and more robust with declared types.

12.1 Type Declarations

The `::` operator can be used to attach type annotations to expressions and variables in programs. There are two primary reasons to do this:

1. As an assertion to help confirm that your program works the way you expect, and
2. To provide extra type information to the compiler, which can then improve performance in some cases.

When appended to an expression computing a value, the `::` operator is read as "is an instance of". It can be used anywhere to assert that the value of the expression on the left is an instance of the type on the right. When the type on the right is concrete, the value on the left must have that type as its implementation – recall that all concrete types are final, so no implementation is a subtype of any other. When the type is abstract, it suffices for the value to be implemented by a concrete type that is a subtype of the abstract type. If the type assertion is not true, an exception is thrown, otherwise, the left-hand value is returned:

```
julia> (1+2)::AbstractFloat
ERROR: TypeError: in typeassert, expected AbstractFloat, got a value of type Int64

julia> (1+2)::Int
3
```

This allows a type assertion to be attached to any expression in-place.

When appended to a variable on the left-hand side of an assignment, or as part of a `local` declaration, the `::` operator means something a bit different: it declares the variable to always have the specified type, like a type declaration in a statically-typed language such as C. Every value assigned to the variable will be converted to the declared type using `convert`:

```
julia> function foo()
           x::Int8 = 100
           x
       end
foo (generic function with 1 method)

julia> x = foo()
100

julia> typeof(x)
Int8
```

This feature is useful for avoiding performance "gotchas" that could occur if one of the assignments to a variable changed its type unexpectedly.

This "declaration" behavior only occurs in specific contexts:

```
local x::Int8  # in a local declaration
x::Int8 = 10   # as the left-hand side of an assignment
```

and applies to the whole current scope, even before the declaration.

As of Julia 1.8, type declarations can now be used in global scope i.e. type annotations can be added to global variables to make accessing them type stable.

```
julia> x::Int = 10
10

julia> x = 3.5
ERROR: InexactError: Int64(3.5)

julia> function foo(y)
           global x = 15.8    # throws an error when foo is called
           return x + y
       end
foo (generic function with 1 method)

julia> foo(10)
ERROR: InexactError: Int64(15.8)
```

Declarations can also be attached to function definitions:

```
function sinc(x)::Float64
    if x == 0
        return 1
    end
    return sin(pi*x)/(pi*x)
end
```

Returning from this function behaves just like an assignment to a variable with a declared type: the value is always converted to `Float64`.

12.2 Abstract Types

Abstract types cannot be instantiated, and serve only as nodes in the type graph, thereby describing sets of related concrete types: those concrete types which are their descendants. We begin with abstract types even though they have no instantiation because they are the backbone of the type system: they form the conceptual hierarchy which makes Julia's type system more than just a collection of object implementations.

Recall that in Integers and Floating-Point Numbers, we introduced a variety of concrete types of numeric values: `Int8`, `UInt8`, `Int16`, `UInt16`, `Int32`, `UInt32`, `Int64`, `UInt64`, `Int128`, `UInt128`, `Float16`, `Float32`, and `Float64`. Although they have different representation sizes, `Int8`, `Int16`, `Int32`, `Int64` and `Int128` all have in common that they are signed integer types. Likewise `UInt8`, `UInt16`, `UInt32`, `UInt64` and `UInt128` are all unsigned integer types, while `Float16`, `Float32` and `Float64` are distinct in being floating-point types rather than integers. It is common for a piece of code to make sense, for example, only if its arguments are some kind of integer, but not really depend on what particular *kind* of integer. For example, the greatest common denominator algorithm works for all kinds of integers, but will not work for floating-point numbers. Abstract types allow the construction of a hierarchy of types, providing a context into which concrete types can fit. This allows you, for example, to easily program to any type that is an integer, without restricting an algorithm to a specific type of integer.

Abstract types are declared using the `abstract type` keyword. The general syntaxes for declaring an abstract type are:

```
abstract type «name» end
abstract type «name» <: «supertype» end
```

The `abstract type` keyword introduces a new abstract type, whose name is given by `«name»`. This name can be optionally followed by `<:` and an already-existing type, indicating that the newly declared abstract type is a subtype of this "parent" type.

When no supertype is given, the default supertype is `Any` – a predefined abstract type that all objects are instances of and all types are subtypes of. In type theory, `Any` is commonly called "top" because it is at the apex of the type graph. Julia also has a predefined abstract "bottom" type, at the nadir of the type graph, which is written as `Union{}`. It is the exact opposite of `Any`: no object is an instance of `Union{}` and all types (including concrete types) are supertypes of `Union{}`.

Let's consider some of the abstract types that make up Julia's numerical hierarchy:

```
abstract type Number end
abstract type Real          <: Number end
abstract type AbstractFloat <: Real end
abstract type Integer       <: Real end
abstract type Signed        <: Integer end
abstract type Unsigned      <: Integer end
```

The `Number` type is a direct child type of `Any`, and `Real` is its child. In turn, `Real` has two children (it has more, but only two are shown here; we'll get to the others later): `Integer` and `AbstractFloat`, separating the world into representations of integers and representations of real numbers. Representations of real numbers include floating-point types, but also include other types, such as rationals. `AbstractFloat` includes only floating-point representations of real numbers. Integers are further subdivided into `Signed` and `Unsigned` varieties.

The `<:` operator in general means "is a subtype of", and, used in declarations like those above, declares the right-hand type to be an immediate supertype of the newly declared type. It can also be used in expressions as a subtype operator which returns `true` when its left operand is a subtype of its right operand:

```
julia> Integer <: Number
true

julia> Integer <: AbstractFloat
false
```

An important use of abstract types is to provide default implementations for concrete types. To give a simple example, consider:

```
function myplus(x,y)
    x+y
end
```

The first thing to note is that the above argument declarations are equivalent to `x::Any` and `y::Any`. When this function is invoked, say as `myplus(2,5)`, the dispatcher chooses the most specific method named `myplus` that matches the given arguments. (See Methods for more information on multiple dispatch.)

Assuming no method more specific than the above is found, Julia next internally defines and compiles a method called `myplus` specifically for two `Int` arguments based on the generic function given above, i.e., it implicitly defines and compiles:

```
function myplus(x::Int,y::Int)
    x+y
end
```

and finally, it invokes this specific method.

Thus, abstract types allow programmers to write generic functions that can later be used as the default method by many combinations of concrete types. Thanks to multiple dispatch, the programmer has full control over whether the default or more specific method is used.

An important point to note is that there is no loss in performance if the programmer relies on a function whose arguments are abstract types, because it is recompiled for each tuple of concrete argument types with which it is invoked. (There may be a performance issue, however, in the case of function arguments that are containers of abstract types; see Performance Tips.)

12.3 Primitive Types

Warning

It is almost always preferable to wrap an existing primitive type in a new composite type than to define your own primitive type.

This functionality exists to allow Julia to bootstrap the standard primitive types that LLVM supports. Once they are defined, there is very little reason to define more.

A primitive type is a concrete type whose data consists of plain old bits. Classic examples of primitive types are integers and floating-point values. Unlike most languages, Julia lets you declare your own primitive types, rather than providing only a fixed set of built-in ones. In fact, the standard primitive types are all defined in the language itself:

```
primitive type Float16 <: AbstractFloat 16 end
primitive type Float32 <: AbstractFloat 32 end
primitive type Float64 <: AbstractFloat 64 end

primitive type Bool <: Integer 8 end
primitive type Char <: AbstractChar 32 end

primitive type Int8    <: Signed   8 end
primitive type UInt8   <: Unsigned 8 end
primitive type Int16   <: Signed   16 end
primitive type UInt16  <: Unsigned 16 end
primitive type Int32   <: Signed   32 end
primitive type UInt32  <: Unsigned 32 end
primitive type Int64   <: Signed   64 end
primitive type UInt64  <: Unsigned 64 end
primitive type Int128  <: Signed   128 end
primitive type UInt128 <: Unsigned 128 end
```

The general syntaxes for declaring a primitive type are:

```
primitive type «name» «bits» end
primitive type «name» <: «supertype» «bits» end
```

The number of bits indicates how much storage the type requires and the name gives the new type a name. A primitive type can optionally be declared to be a subtype of some supertype. If a supertype is omitted, then the type defaults to having Any as its immediate supertype. The declaration of `Bool` above therefore means that a boolean value takes eight bits to store, and has `Integer` as its immediate supertype. Currently, only sizes that are multiples of 8 bits are supported and you are more likely to experience bugs with sizes other than those used above. Therefore, boolean values, although they really need just a single bit, cannot be declared to be any smaller than eight bits.

The types `Bool`, `Int8` and `UInt8` all have identical representations: they are eight-bit chunks of memory. Since Julia's type system is nominative, however, they are not interchangeable despite having identical structure. A fundamental difference between them is that they have different supertypes: `Bool`'s direct supertype is `Integer`, `Int8`'s is `Signed`, and `UInt8`'s is `Unsigned`. All other differences between `Bool`, `Int8`, and `UInt8` are matters of behavior – the way functions are defined to act when given objects of these types as arguments. This is why a nominative type system is necessary: if structure determined type, which in turn dictates behavior, then it would be impossible to make `Bool` behave any differently than `Int8` or `UInt8`.

12.4 Composite Types

Composite types are called records, structs, or objects in various languages. A composite type is a collection of named fields, an instance of which can be treated as a single value. In many languages, composite types are the only kind of user-definable type, and they are by far the most commonly used user-defined type in Julia as well.

In mainstream object oriented languages, such as C++, Java, Python and Ruby, composite types also have named functions associated with them, and the combination is called an "object". In purer object-oriented languages, such as Ruby or Smalltalk, all values are objects whether they are composites or not. In less pure object oriented languages, including C++ and Java, some values, such as integers and floating-point values, are not objects, while instances of user-defined composite types are true objects with associated methods. In Julia, all values are objects, but functions are not bundled with the objects they operate on. This is necessary since Julia chooses which method of a function to use by multiple dispatch, meaning that the types of *all* of a function's arguments are considered when selecting a method, rather than just the first one (see Methods for more information on methods and dispatch). Thus, it would be inappropriate for functions to "belong" to only their first argument. Organizing methods into function objects rather than having named bags of methods "inside" each object ends up being a highly beneficial aspect of the language design.

Composite types are introduced with the `struct` keyword followed by a block of field names, optionally annotated with types using the `::` operator:

```
julia> struct Foo
           bar
           baz::Int
           qux::Float64
       end
```

Fields with no type annotation default to Any, and can accordingly hold any type of value.

New objects of type `Foo` are created by applying the `Foo` type object like a function to values for its fields:

```
julia> foo = Foo("Hello, world.", 23, 1.5)
Foo("Hello, world.", 23, 1.5)

julia> typeof(foo)
Foo
```

When a type is applied like a function it is called a *constructor*. Two constructors are generated automatically (these are called *default constructors*). One accepts any arguments and calls `convert` to convert them to the types of the fields, and the other accepts arguments that match the field types exactly. The reason both of these are generated is that this makes it easier to add new definitions without inadvertently replacing a default constructor.

Since the `bar` field is unconstrained in type, any value will do. However, the value for `baz` must be convertible to `Int`:

```
julia> Foo((), 23.5, 1)
ERROR: InexactError: Int64(23.5)
Stacktrace:
[...]
```

You may find a list of field names using the `fieldnames` function.

```
julia> fieldnames(Foo)
(:bar, :baz, :qux)
```

You can access the field values of a composite object using the traditional `foo.bar` notation:

```
julia> foo.bar
"Hello, world."

julia> foo.baz
23

julia> foo.qux
1.5
```

Composite objects declared with `struct` are *immutable*; they cannot be modified after construction. This may seem odd at first, but it has several advantages:

- It can be more efficient. Some structs can be packed efficiently into arrays, and in some cases the compiler is able to avoid allocating immutable objects entirely.
- It is not possible to violate the invariants provided by the type's constructors.
- Code using immutable objects can be easier to reason about.

An immutable object might contain mutable objects, such as arrays, as fields. Those contained objects will remain mutable; only the fields of the immutable object itself cannot be changed to point to different objects.

Where required, mutable composite objects can be declared with the keyword `mutable struct`, to be discussed in the next section.

If all the fields of an immutable structure are indistinguishable (===) then two immutable values containing those fields are also indistinguishable:

```
julia> struct X
           a::Int
           b::Float64
       end

julia> X(1, 2) === X(1, 2)
true
```

There is much more to say about how instances of composite types are created, but that discussion depends on both Parametric Types and on Methods, and is sufficiently important to be addressed in its own section: Constructors.

For many user-defined types X, you may want to define a method `Base.broadcastable(x::X) = Ref(x)` so that instances of that type act as 0-dimensional "scalars" for broadcasting.

12.5 Mutable Composite Types

If a composite type is declared with `mutable struct` instead of `struct`, then instances of it can be modified:

```
julia> mutable struct Bar
           baz
           qux::Float64
       end

julia> bar = Bar("Hello", 1.5);

julia> bar.qux = 2.0
2.0

julia> bar.baz = 1//2
1//2
```

An extra interface between the fields and the user can be provided through Instance Properties. This grants more control on what can be accessed and modified using the `bar.baz` notation.

In order to support mutation, such objects are generally allocated on the heap, and have stable memory addresses. A mutable object is like a little container that might hold different values over time, and so can only be reliably identified with its address. In contrast, an instance of an immutable type is associated with specific field values — the field values alone tell you everything about the object. In deciding whether to make a type mutable, ask whether two instances with the same field values would be considered identical, or if they might need to change independently over time. If they would be considered identical, the type should probably be immutable.

To recap, two essential properties define immutability in Julia:

- It is not permitted to modify the value of an immutable type.
  - For bits types this means that the bit pattern of a value once set will never change and that value is the identity of a bits type.
  - For composite types, this means that the identity of the values of its fields will never change. When the fields are bits types, that means their bits will never change, for fields whose values are mutable types like arrays, that means the fields will always refer to the same mutable value even though that mutable value's content may itself be modified.
- An object with an immutable type may be copied freely by the compiler since its immutability makes it impossible to programmatically distinguish between the original object and a copy.
  - In particular, this means that small enough immutable values like integers and floats are typically passed to functions in registers (or stack allocated).
  - Mutable values, on the other hand are heap-allocated and passed to functions as pointers to heap-allocated values except in cases where the compiler is sure that there's no way to tell that this is not what is happening.

In cases where one or more fields of an otherwise mutable struct is known to be immutable, one can declare these fields as such using `const` as shown below. This enables some, but not all of the optimizations of immutable structs, and can be used to enforce invariants on the particular fields marked as `const`.

Julia 1.8

`const` annotating fields of mutable structs requires at least Julia 1.8.

```
julia> mutable struct Baz
           a::Int
           const b::Float64
       end

julia> baz = Baz(1, 1.5);

julia> baz.a = 2
2

julia> baz.b = 2.0
ERROR: setfield!: const field .b of type Baz cannot be changed
[...]
```

12.6 Declared Types

The three kinds of types (abstract, primitive, composite) discussed in the previous sections are actually all closely related. They share the same key properties:

- They are explicitly declared.
- They have names.
- They have explicitly declared supertypes.
- They may have parameters.

Because of these shared properties, these types are internally represented as instances of the same concept, `DataType`, which is the type of any of these types:

```
julia> typeof(Real)
DataType

julia> typeof(Int)
DataType
```

A `DataType` may be abstract or concrete. If it is concrete, it has a specified size, storage layout, and (optionally) field names. Thus a primitive type is a `DataType` with nonzero size, but no field names. A composite type is a `DataType` that has field names or is empty (zero size).

Every concrete value in the system is an instance of some `DataType`.

12.7 Type Unions

A type union is a special abstract type which includes as objects all instances of any of its argument types, constructed using the special `Union` keyword:

```
julia> IntOrString = Union{Int,AbstractString}
Union{Int64, AbstractString}
```

```
julia> 1 :: IntOrString
1

julia> "Hello!" :: IntOrString
"Hello!"

julia> 1.0 :: IntOrString
ERROR: TypeError: in typeassert, expected Union{Int64, AbstractString}, got a value of type
↪  Float64
```

The compilers for many languages have an internal union construct for reasoning about types; Julia simply exposes it to the programmer. The Julia compiler is able to generate efficient code in the presence of `Union` types with a small number of types [1], by generating specialized code in separate branches for each possible type.

A particularly useful case of a `Union` type is `Union{T, Nothing}`, where T can be any type and `Nothing` is the singleton type whose only instance is the object `nothing`. This pattern is the Julia equivalent of `Nullable`, `Option` or `Maybe` types in other languages. Declaring a function argument or a field as `Union{T, Nothing}` allows setting it either to a value of type T, or to `nothing` to indicate that there is no value. See this FAQ entry for more information.

12.8 Parametric Types

An important and powerful feature of Julia's type system is that it is parametric: types can take parameters, so that type declarations actually introduce a whole family of new types – one for each possible combination of parameter values. There are many languages that support some version of generic programming, wherein data structures and algorithms to manipulate them may be specified without specifying the exact types involved. For example, some form of generic programming exists in ML, Haskell, Ada, Eiffel, C++, Java, C#, F#, and Scala, just to name a few. Some of these languages support true parametric polymorphism (e.g. ML, Haskell, Scala), while others support ad-hoc, template-based styles of generic programming (e.g. C++, Java). With so many different varieties of generic programming and parametric types in various languages, we won't even attempt to compare Julia's parametric types to other languages, but will instead focus on explaining Julia's system in its own right. We will note, however, that because Julia is a dynamically typed language and doesn't need to make all type decisions at compile time, many traditional difficulties encountered in static parametric type systems can be relatively easily handled.

All declared types (the `DataType` variety) can be parameterized, with the same syntax in each case. We will discuss them in the following order: first, parametric composite types, then parametric abstract types, and finally parametric primitive types.

Parametric Composite Types

Type parameters are introduced immediately after the type name, surrounded by curly braces:

```
julia> struct Point{T}
           x::T
           y::T
       end
```

This declaration defines a new parametric type, `Point{T}`, holding two "coordinates" of type T. What, one may ask, is T? Well, that's precisely the point of parametric types: it can be any type at all (or a value

of any bits type, actually, although here it's clearly used as a type). `Point{Float64}` is a concrete type equivalent to the type defined by replacing T in the definition of `Point` with `Float64`. Thus, this single declaration actually declares an unlimited number of types: `Point{Float64}`, `Point{AbstractString}`, `Point{Int64}`, etc. Each of these is now a usable concrete type:

```
julia> Point{Float64}
Point{Float64}

julia> Point{AbstractString}
Point{AbstractString}
```

The type `Point{Float64}` is a point whose coordinates are 64-bit floating-point values, while the type `Point{AbstractString}` is a "point" whose "coordinates" are string objects (see Strings).

`Point` itself is also a valid type object, containing all instances `Point{Float64}`, `Point{AbstractString}`, etc. as subtypes:

```
julia> Point{Float64} <: Point
true

julia> Point{AbstractString} <: Point
true
```

Other types, of course, are not subtypes of it:

```
julia> Float64 <: Point
false

julia> AbstractString <: Point
false
```

Concrete `Point` types with different values of T are never subtypes of each other:

```
julia> Point{Float64} <: Point{Int64}
false

julia> Point{Float64} <: Point{Real}
false
```

Warning

This last point is *very* important: even though `Float64 <: Real` we **DO NOT** have `Point{Float64} <: Point{Real}`.

In other words, in the parlance of type theory, Julia's type parameters are *invariant*, rather than being covariant (or even contravariant). This is for practical reasons: while any instance of `Point{Float64}` may conceptually be like an instance of `Point{Real}` as well, the two types have different representations in memory:

- An instance of `Point{Float64}` can be represented compactly and efficiently as an immediate pair of 64-bit values;
- An instance of `Point{Real}` must be able to hold any pair of instances of `Real`. Since objects that are instances of `Real` can be of arbitrary size and structure, in practice an instance of `Point{Real}` must be represented as a pair of pointers to individually allocated `Real` objects.

The efficiency gained by being able to store `Point{Float64}` objects with immediate values is magnified enormously in the case of arrays: an `Array{Float64}` can be stored as a contiguous memory block of 64-bit floating-point values, whereas an `Array{Real}` must be an array of pointers to individually allocated `Real` objects – which may well be boxed 64-bit floating-point values, but also might be arbitrarily large, complex objects, which are declared to be implementations of the `Real` abstract type.

Since `Point{Float64}` is not a subtype of `Point{Real}`, the following method can't be applied to arguments of type `Point{Float64}`:

```
function norm(p::Point{Real})
    sqrt(p.x^2 + p.y^2)
end
```

A correct way to define a method that accepts all arguments of type `Point{T}` where T is a subtype of `Real` is:

```
function norm(p::Point{<:Real})
    sqrt(p.x^2 + p.y^2)
end
```

(Equivalently, one could define `function norm(p::Point{T} where T<:Real)` or `function norm(p::Point{T}) where T<:Real`; see UnionAll Types.)

More examples will be discussed later in Methods.

How does one construct a `Point` object? It is possible to define custom constructors for composite types, which will be discussed in detail in Constructors, but in the absence of any special constructor declarations, there are two default ways of creating new composite objects, one in which the type parameters are explicitly given and the other in which they are implied by the arguments to the object constructor.

Since the type `Point{Float64}` is a concrete type equivalent to `Point` declared with `Float64` in place of T, it can be applied as a constructor accordingly:

```
julia> p = Point{Float64}(1.0, 2.0)
Point{Float64}(1.0, 2.0)

julia> typeof(p)
Point{Float64}
```

For the default constructor, exactly one argument must be supplied for each field:

```
julia> Point{Float64}(1.0)
ERROR: MethodError: no method matching Point{Float64}(::Float64)
The type `Point{Float64}` exists, but no method is defined for this combination of argument types
↪  when trying to construct it.
[...]

julia> Point{Float64}(1.0, 2.0, 3.0)
ERROR: MethodError: no method matching Point{Float64}(::Float64, ::Float64, ::Float64)
The type `Point{Float64}` exists, but no method is defined for this combination of argument types
↪  when trying to construct it.
[...]
```

Only one default constructor is generated for parametric types, since overriding it is not possible. This constructor accepts any arguments and converts them to the field types.

In many cases, it is redundant to provide the type of `Point` object one wants to construct, since the types of arguments to the constructor call already implicitly provide type information. For that reason, you can also apply `Point` itself as a constructor, provided that the implied value of the parameter type T is unambiguous:

```
julia> p1 = Point(1.0,2.0)
Point{Float64}(1.0, 2.0)

julia> typeof(p1)
Point{Float64}

julia> p2 = Point(1,2)
Point{Int64}(1, 2)

julia> typeof(p2)
Point{Int64}
```

In the case of `Point`, the type of T is unambiguously implied if and only if the two arguments to `Point` have the same type. When this isn't the case, the constructor will fail with a `MethodError`:

```
julia> Point(1,2.5)
ERROR: MethodError: no method matching Point(::Int64, ::Float64)
The type `Point` exists, but no method is defined for this combination of argument types when
↪  trying to construct it.

Closest candidates are:
  Point(::T, !Matched::T) where T
   @ Main none:2

Stacktrace:
[...]
```

Constructor methods to appropriately handle such mixed cases can be defined, but that will not be discussed until later on in Constructors.

Parametric Abstract Types

Parametric abstract type declarations declare a collection of abstract types, in much the same way:

```
julia> abstract type Pointy{T} end
```

With this declaration, `Pointy{T}` is a distinct abstract type for each type or integer value of T. As with parametric composite types, each such instance is a subtype of `Pointy`:

```
julia> Pointy{Int64} <: Pointy
true

julia> Pointy{1} <: Pointy
true
```

Parametric abstract types are invariant, much as parametric composite types are:

```
julia> Pointy{Float64} <: Pointy{Real}
false

julia> Pointy{Real} <: Pointy{Float64}
false
```

The notation `Pointy{<:Real}` can be used to express the Julia analogue of a *covariant* type, while `Pointy{>:Int}` the analogue of a *contravariant* type, but technically these represent *sets* of types (see UnionAll Types).

```
julia> Pointy{Float64} <: Pointy{<:Real}
true

julia> Pointy{Real} <: Pointy{>:Int}
true
```

Much as plain old abstract types serve to create a useful hierarchy of types over concrete types, parametric abstract types serve the same purpose with respect to parametric composite types. We could, for example, have declared `Point{T}` to be a subtype of `Pointy{T}` as follows:

```
julia> struct Point{T} <: Pointy{T}
           x::T
           y::T
       end
```

Given such a declaration, for each choice of T, we have `Point{T}` as a subtype of `Pointy{T}`:

```
julia> Point{Float64} <: Pointy{Float64}
true

julia> Point{Real} <: Pointy{Real}
true

julia> Point{AbstractString} <: Pointy{AbstractString}
true
```

This relationship is also invariant:

```
julia> Point{Float64} <: Pointy{Real}
false

julia> Point{Float64} <: Pointy{<:Real}
true
```

What purpose do parametric abstract types like `Pointy` serve? Consider if we create a point-like implementation that only requires a single coordinate because the point is on the diagonal line $x = y$:

```
julia> struct DiagPoint{T} <: Pointy{T}
           x::T
       end
```

Now both `Point{Float64}` and `DiagPoint{Float64}` are implementations of the `Pointy{Float64}` abstraction, and similarly for every other possible choice of type T. This allows programming to a common interface shared by all `Pointy` objects, implemented for both `Point` and `DiagPoint`. This cannot be fully demonstrated, however, until we have introduced methods and dispatch in the next section, Methods.

There are situations where it may not make sense for type parameters to range freely over all possible types. In such situations, one can constrain the range of T like so:

```
julia> abstract type Pointy{T<:Real} end
```

With such a declaration, it is acceptable to use any type that is a subtype of `Real` in place of T, but not types that are not subtypes of `Real`:

```
julia> Pointy{Float64}
Pointy{Float64}

julia> Pointy{Real}
Pointy{Real}

julia> Pointy{AbstractString}
ERROR: TypeError: in Pointy, in T, expected T<:Real, got Type{AbstractString}

julia> Pointy{1}
ERROR: TypeError: in Pointy, in T, expected T<:Real, got a value of type Int64
```

Type parameters for parametric composite types can be restricted in the same manner:

```
struct Point{T<:Real} <: Pointy{T}
    x::T
    y::T
end
```

To give a real-world example of how all this parametric type machinery can be useful, here is the actual definition of Julia's `Rational` immutable type (except that we omit the constructor here for simplicity), representing an exact ratio of integers:

```
struct Rational{T<:Integer} <: Real
    num::T
    den::T
end
```

It only makes sense to take ratios of integer values, so the parameter type T is restricted to being a subtype of `Integer`, and a ratio of integers represents a value on the real number line, so any `Rational` is an instance of the `Real` abstraction.

Tuple Types

Tuples are an abstraction of the arguments of a function – without the function itself. The salient aspects of a function's arguments are their order and their types. Therefore a tuple type is similar to a parameterized immutable type where each parameter is the type of one field. For example, a 2-element tuple type resembles the following immutable type:

```
struct Tuple2{A,B}
    a::A
    b::B
end
```

However, there are three key differences:

- Tuple types may have any number of parameters.
- Tuple types are *covariant* in their parameters: `Tuple{Int}` is a subtype of `Tuple{Any}`. Therefore `Tuple{Any}` is considered an abstract type, and tuple types are only concrete if their parameters are.
- Tuples do not have field names; fields are only accessed by index.

Tuple values are written with parentheses and commas. When a tuple is constructed, an appropriate tuple type is generated on demand:

```
julia> typeof((1,"foo",2.5))
Tuple{Int64, String, Float64}
```

Note the implications of covariance:

```
julia> Tuple{Int,AbstractString} <: Tuple{Real,Any}
true

julia> Tuple{Int,AbstractString} <: Tuple{Real,Real}
false

julia> Tuple{Int,AbstractString} <: Tuple{Real,}
false
```

Intuitively, this corresponds to the type of a function's arguments being a subtype of the function's signature (when the signature matches).

Vararg Tuple Types

The last parameter of a tuple type can be the special value `Vararg`, which denotes any number of trailing elements:

```
julia> mytupletype = Tuple{AbstractString,Vararg{Int}}
Tuple{AbstractString, Vararg{Int64}}

julia> isa(("1",), mytupletype)
true

julia> isa(("1",1), mytupletype)
true

julia> isa(("1",1,2), mytupletype)
true

julia> isa(("1",1,2,3.0), mytupletype)
false
```

Moreover `Vararg{T}` corresponds to zero or more elements of type `T`. Vararg tuple types are used to represent the arguments accepted by varargs methods (see Varargs Functions).

The special value `Vararg{T,N}` (when used as the last parameter of a tuple type) corresponds to exactly `N` elements of type `T`. `NTuple{N,T}` is a convenient alias for `Tuple{Vararg{T,N}}`, i.e. a tuple type containing exactly `N` elements of type `T`.

Named Tuple Types

Named tuples are instances of the `NamedTuple` type, which has two parameters: a tuple of symbols giving the field names, and a tuple type giving the field types. For convenience, `NamedTuple` types are printed using the `@NamedTuple` macro which provides a convenient `struct`-like syntax for declaring these types via `key::Type` declarations, where an omitted `::Type` corresponds to `::Any`.

```
julia> typeof((a=1,b="hello")) # prints in macro form
@NamedTuple{a::Int64, b::String}

julia> NamedTuple{(:a, :b), Tuple{Int64, String}} # long form of the type
@NamedTuple{a::Int64, b::String}
```

The `begin ... end` form of the `@NamedTuple` macro allows the declarations to be split across multiple lines (similar to a struct declaration), but is otherwise equivalent:

```
julia> @NamedTuple begin
           a::Int
           b::String
       end
@NamedTuple{a::Int64, b::String}
```

A `NamedTuple` type can be used as a constructor, accepting a single tuple argument. The constructed `NamedTuple` type can be either a concrete type, with both parameters specified, or a type that specifies only field names:

```
julia> @NamedTuple{a::Float32,b::String}((1, ""))
(a = 1.0f0, b = "")

julia> NamedTuple{(:a, :b)}((1, ""))
(a = 1, b = "")
```

If field types are specified, the arguments are converted. Otherwise the types of the arguments are used directly.

Parametric Primitive Types

Primitive types can also be declared parametrically. For example, pointers are represented as primitive types which would be declared in Julia like this:

```
# 32-bit system:
primitive type Ptr{T} 32 end

# 64-bit system:
primitive type Ptr{T} 64 end
```

The slightly odd feature of these declarations as compared to typical parametric composite types, is that the type parameter T is not used in the definition of the type itself – it is just an abstract tag, essentially defining an entire family of types with identical structure, differentiated only by their type parameter. Thus, `Ptr{Float64}` and `Ptr{Int64}` are distinct types, even though they have identical representations. And of course, all specific pointer types are subtypes of the umbrella `Ptr` type:

```
julia> Ptr{Float64} <: Ptr
true

julia> Ptr{Int64} <: Ptr
true
```

12.9 UnionAll Types

We have said that a parametric type like `Ptr` acts as a supertype of all its instances (`Ptr{Int64}` etc.). How does this work? `Ptr` itself cannot be a normal data type, since without knowing the type of the referenced data the type clearly cannot be used for memory operations. The answer is that `Ptr` (or other parametric types like `Array`) is a different kind of type called a `UnionAll` type. Such a type expresses the *iterated union* of types for all values of some parameter.

`UnionAll` types are usually written using the keyword `where`. For example `Ptr` could be more accurately written as `Ptr{T} where T`, meaning all values whose type is `Ptr{T}` for some value of `T`. In this context, the parameter `T` is also often called a "type variable" since it is like a variable that ranges over types. Each `where` introduces a single type variable, so these expressions are nested for types with multiple parameters, for example `Array{T,N} where N where T`.

The type application syntax `A{B,C}` requires `A` to be a `UnionAll` type, and first substitutes `B` for the outermost type variable in `A`. The result is expected to be another `UnionAll` type, into which `C` is then substituted. So `A{B,C}` is equivalent to `A{B}{C}`. This explains why it is possible to partially instantiate a type, as in `Array{Float64}`: the first parameter value has been fixed, but the second still ranges over all possible

values. Using explicit `where` syntax, any subset of parameters can be fixed. For example, the type of all 1-dimensional arrays can be written as `Array{T,1} where T`.

Type variables can be restricted with subtype relations. `Array{T} where T<:Integer` refers to all arrays whose element type is some kind of `Integer`. The syntax `Array{<:Integer}` is a convenient shorthand for `Array{T} where T<:Integer`. Type variables can have both lower and upper bounds. `Array{T} where Int<:T<:Number` refers to all arrays of `Number`s that are able to contain `Int`s (since T must be at least as big as `Int`). The syntax `where T>:Int` also works to specify only the lower bound of a type variable, and `Array{>:Int}` is equivalent to `Array{T} where T>:Int`.

Since `where` expressions nest, type variable bounds can refer to outer type variables. For example `Tuple{T,Array{S}} where S<:AbstractArray{T} where T<:Real` refers to 2-tuples whose first element is some `Real`, and whose second element is an `Array` of any kind of array whose element type contains the type of the first tuple element.

The `where` keyword itself can be nested inside a more complex declaration. For example, consider the two types created by the following declarations:

```
julia> const T1 = Array{Array{T, 1} where T, 1}
Vector{Vector} (alias for Array{Array{T, 1} where T, 1})

julia> const T2 = Array{Array{T, 1}, 1} where T
Array{Vector{T}, 1} where T
```

Type T1 defines a 1-dimensional array of 1-dimensional arrays; each of the inner arrays consists of objects of the same type, but this type may vary from one inner array to the next. On the other hand, type T2 defines a 1-dimensional array of 1-dimensional arrays all of whose inner arrays must have the same type. Note that T2 is an abstract type, e.g., `Array{Array{Int,1},1} <: T2`, whereas T1 is a concrete type. As a consequence, T1 can be constructed with a zero-argument constructor `a=T1()` but T2 cannot.

There is a convenient syntax for naming such types, similar to the short form of function definition syntax:

```
Vector{T} = Array{T, 1}
```

This is equivalent to `const Vector = Array{T,1} where T`. Writing `Vector{Float64}` is equivalent to writing `Array{Float64,1}`, and the umbrella type `Vector` has as instances all `Array` objects where the second parameter – the number of array dimensions – is 1, regardless of what the element type is. In languages where parametric types must always be specified in full, this is not especially helpful, but in Julia, this allows one to write just `Vector` for the abstract type including all one-dimensional dense arrays of any element type.

12.10 Singleton types

Immutable composite types with no fields are called *singletons*. Formally, if

1. T is an immutable composite type (i.e. defined with `struct`),
2. `a isa T && b isa T` implies `a === b`,

then T is a singleton type.[2] `Base.issingletontype` can be used to check if a type is a singleton type. Abstract types cannot be singleton types by construction.

From the definition, it follows that there can be only one instance of such types:

```
julia> struct NoFields
       end

julia> NoFields() === NoFields()
true

julia> Base.issingletontype(NoFields)
true
```

The `===` function confirms that the constructed instances of `NoFields` are actually one and the same.

Parametric types can be singleton types when the above condition holds. For example,

```
julia> struct NoFieldsParam{T}
       end

julia> Base.issingletontype(NoFieldsParam) # Can't be a singleton type ...
false

julia> NoFieldsParam{Int}() isa NoFieldsParam # ... because it has ...
true

julia> NoFieldsParam{Bool}() isa NoFieldsParam # ... multiple instances.
true

julia> Base.issingletontype(NoFieldsParam{Int}) # Parametrized, it is a singleton.
true

julia> NoFieldsParam{Int}() === NoFieldsParam{Int}()
true
```

12.11 Types of functions

Each function has its own type, which is a subtype of `Function`.

```
julia> foo41(x) = x + 1
foo41 (generic function with 1 method)

julia> typeof(foo41)
typeof(foo41) (singleton type of function foo41, subtype of Function)
```

Note how `typeof(foo41)` prints as itself. This is merely a convention for printing, as it is a first-class object that can be used like any other value:

```
julia> T = typeof(foo41)
typeof(foo41) (singleton type of function foo41, subtype of Function)
```

```
julia> T <: Function
true
```

Types of functions defined at top-level are singletons. When necessary, you can compare them with `===`.

Closures also have their own type, which is usually printed with names that end in `#<number>`. Names and types for functions defined at different locations are distinct, but not guaranteed to be printed the same way across sessions.

```
julia> typeof(x -> x + 1)
var"#9#10"
```

Types of closures are not necessarily singletons.

```
julia> addy(y) = x -> x + y
addy (generic function with 1 method)

julia> typeof(addy(1)) === typeof(addy(2))
true

julia> addy(1) === addy(2)
false

julia> Base.issingletontype(typeof(addy(1)))
false
```

12.12 Type{T} type selectors

For each type T, Type{T} is an abstract parametric type whose only instance is the object T. Until we discuss Parametric Methods and conversions, it is difficult to explain the utility of this construct, but in short, it allows one to specialize function behavior on specific types as *values*. This is useful for writing methods (especially parametric ones) whose behavior depends on a type that is given as an explicit argument rather than implied by the type of one of its arguments.

Since the definition is a little difficult to parse, let's look at some examples:

```
julia> isa(Float64, Type{Float64})
true

julia> isa(Real, Type{Float64})
false

julia> isa(Real, Type{Real})
true

julia> isa(Float64, Type{Real})
false
```

In other words, `isa(A, Type{B})` is true if and only if A and B are the same object and that object is a type.

In particular, since parametric types are invariant, we have

```
julia> struct TypeParamExample{T}
           x::T
       end

julia> TypeParamExample isa Type{TypeParamExample}
true

julia> TypeParamExample{Int} isa Type{TypeParamExample}
false

julia> TypeParamExample{Int} isa Type{TypeParamExample{Int}}
true
```

Without the parameter, Type is simply an abstract type which has all type objects as its instances:

```
julia> isa(Type{Float64}, Type)
true

julia> isa(Float64, Type)
true

julia> isa(Real, Type)
true
```

Any object that is not a type is not an instance of Type:

```
julia> isa(1, Type)
false

julia> isa("foo", Type)
false
```

While Type is part of Julia's type hierarchy like any other abstract parametric type, it is not commonly used outside method signatures except in some special cases. Another important use case for Type is sharpening field types which would otherwise be captured less precisely, e.g. as DataType in the example below where the default constructor could lead to performance problems in code relying on the precise wrapped type (similarly to abstract type parameters).

```
julia> struct WrapType{T}
       value::T
       end

julia> WrapType(Float64) # default constructor, note DataType
WrapType{DataType}(Float64)

julia> WrapType(::Type{T}) where T = WrapType{Type{T}}(T)
WrapType

julia> WrapType(Float64) # sharpened constructor, note more precise Type{Float64}
WrapType{Type{Float64}}(Float64)
```

This behavior of Type{Float64} is an example of an abstract type subtyping a concrete type (here DataType).

12.13 Type Aliases

Sometimes it is convenient to introduce a new name for an already expressible type. This can be done with a simple assignment statement. For example, `UInt` is aliased to either `UInt32` or `UInt64` as is appropriate for the size of pointers on the system:

```
# 32-bit system:
julia> UInt
UInt32

# 64-bit system:
julia> UInt
UInt64
```

This is accomplished via the following code in `base/boot.jl`:

```
if Int === Int64
    const UInt = UInt64
else
    const UInt = UInt32
end
```

Of course, this depends on what `Int` is aliased to – but that is predefined to be the correct type – either `Int32` or `Int64`.

(Note that unlike `Int`, `Float` does not exist as a type alias for a specific sized `AbstractFloat`. Unlike with integer registers, where the size of `Int` reflects the size of a native pointer on that machine, the floating point register sizes are specified by the IEEE-754 standard.)

Type aliases may be parametrized:

```
julia> const Family{T} = Set{T}
Set

julia> Family{Char} === Set{Char}
true
```

12.14 Operations on Types

Since types in Julia are themselves objects, ordinary functions can operate on them. Some functions that are particularly useful for working with or exploring types have already been introduced, such as the `<:` operator, which indicates whether its left hand operand is a subtype of its right hand operand.

The `isa` function tests if an object is of a given type and returns true or false:

```
julia> isa(1, Int)
true

julia> isa(1, AbstractFloat)
false
```

The `typeof` function, already used throughout the manual in examples, returns the type of its argument. Since, as noted above, types are objects, they also have types, and we can ask what their types are:

```
julia> typeof(Rational{Int})
DataType

julia> typeof(Union{Real,String})
Union
```

What if we repeat the process? What is the type of a type of a type? As it happens, types are all composite values and thus all have a type of `DataType`:

```
julia> typeof(DataType)
DataType

julia> typeof(Union)
DataType
```

`DataType` is its own type.

Another operation that applies to some types is `supertype`, which reveals a type's supertype. Only declared types (`DataType`) have unambiguous supertypes:

```
julia> supertype(Float64)
AbstractFloat

julia> supertype(Number)
Any

julia> supertype(AbstractString)
Any

julia> supertype(Any)
Any
```

If you apply `supertype` to other type objects (or non-type objects), a `MethodError` is raised:

```
julia> supertype(Union{Float64,Int64})
ERROR: MethodError: no method matching supertype(::Type{Union{Float64, Int64}})
The function `supertype` exists, but no method is defined for this combination of argument types.

Closest candidates are:
[...]
```

12.15 Custom pretty-printing

Often, one wants to customize how instances of a type are displayed. This is accomplished by overloading the `show` function. For example, suppose we define a type to represent complex numbers in polar form:

```
julia> struct Polar{T<:Real} <: Number
           r::T
           θ::T
       end

julia> Polar(r::Real,θ::Real) = Polar(promote(r,θ)...)
Polar
```

Here, we've added a custom constructor function so that it can take arguments of different `Real` types and promote them to a common type (see Constructors and Conversion and Promotion). (Of course, we would have to define lots of other methods, too, to make it act like a `Number`, e.g. `+`, `*`, `one`, `zero`, promotion rules and so on.) By default, instances of this type display rather simply, with information about the type name and the field values, as e.g. `Polar{Float64}(3.0,4.0)`.

If we want it to display instead as `3.0 * exp(4.0im)`, we would define the following method to print the object to a given output object `io` (representing a file, terminal, buffer, etcetera; see Networking and Streams):

```
julia> Base.show(io::IO, z::Polar) = print(io, z.r, " * exp(", z.θ, "im)")
```

More fine-grained control over display of `Polar` objects is possible. In particular, sometimes one wants both a verbose multi-line printing format, used for displaying a single object in the REPL and other interactive environments, and also a more compact single-line format used for `print` or for displaying the object as part of another object (e.g. in an array). Although by default the `show(io, z)` function is called in both cases, you can define a *different* multi-line format for displaying an object by overloading a three-argument form of `show` that takes the `text/plain` MIME type as its second argument (see Multimedia I/O), for example:

```
julia> Base.show(io::IO, ::MIME"text/plain", z::Polar{T}) where{T} =
           print(io, "Polar{$T} complex number:\n   ", z)
```

(Note that `print(..., z)` here will call the 2-argument `show(io, z)` method.) This results in:

```
julia> Polar(3, 4.0)
Polar{Float64} complex number:
   3.0 * exp(4.0im)

julia> [Polar(3, 4.0), Polar(4.0,5.3)]
2-element Vector{Polar{Float64}}:
 3.0 * exp(4.0im)
 4.0 * exp(5.3im)
```

where the single-line `show(io, z)` form is still used for an array of `Polar` values. Technically, the REPL calls `display(z)` to display the result `z` of executing a line, which defaults to `show(io, MIME("text/plain"), z)` (where `io` is an `IOContext` wrapper around `stdout`), which in turn defaults to `show(io, z)`, but you should *not* define new `display` methods unless you are defining a new multimedia display handler (see Multimedia I/O).

Moreover, you can also define `show` methods for other MIME types in order to enable richer display (HTML, images, etcetera) of objects in environments that support this (e.g. IJulia). For example, we can define formatted HTML display of `Polar` objects, with superscripts and italics, via:

```
julia> Base.show(io::IO, ::MIME"text/html", z::Polar{T}) where {T} =
           println(io, "<code>Polar{$T}</code> complex number: ",
                   z.r, " <i>e</i><sup>", z.θ, " <i>i</i></sup>")
```

A `Polar` object will then display automatically using HTML in an environment that supports HTML display, but you can call show manually to get HTML output if you want:

```
julia> show(stdout, "text/html", Polar(3.0,4.0))
<code>Polar{Float64}</code> complex number: 3.0 <i>e</i><sup>4.0 <i>i</i></sup>
```

As a rule of thumb, the single-line show method should print a valid Julia expression for creating the shown object. When this show method contains infix operators, such as the multiplication operator (`*`) in our single-line show method for `Polar` above, it may not parse correctly when printed as part of another object. To see this, consider the expression object (see Program representation) which takes the square of a specific instance of our `Polar` type:

```
julia> a = Polar(3, 4.0)
Polar{Float64} complex number:
   3.0 * exp(4.0im)

julia> print(:($a^2))
3.0 * exp(4.0im) ^ 2
```

Because the operator `^` has higher precedence than `*` (see Operator Precedence and Associativity), this output does not faithfully represent the expression `a ^ 2` which should be equal to `(3.0 * exp(4.0im)) ^ 2`. To solve this issue, we must make a custom method for `Base.show_unquoted(io::IO, z::Polar, indent::Int, precedence::Int)`, which is called internally by the expression object when printing:

```
julia> function Base.show_unquoted(io::IO, z::Polar, ::Int, precedence::Int)
           if Base.operator_precedence(:*) <= precedence
               print(io, "(")
               show(io, z)
               print(io, ")")
           else
               show(io, z)
           end
       end

julia> :($a^2)
:((3.0 * exp(4.0im)) ^ 2)
```

The method defined above adds parentheses around the call to show when the precedence of the calling operator is higher than or equal to the precedence of multiplication. This check allows expressions which parse correctly without the parentheses (such as `:($a + 2)` and `:($a == 2)`) to omit them when printing:

```
julia> :($a + 2)
:(3.0 * exp(4.0im) + 2)

julia> :($a == 2)
:(3.0 * exp(4.0im) == 2)
```

In some cases, it is useful to adjust the behavior of `show` methods depending on the context. This can be achieved via the `IOContext` type, which allows passing contextual properties together with a wrapped IO stream. For example, we can build a shorter representation in our `show` method when the `:compact` property is set to `true`, falling back to the long representation if the property is `false` or absent:

```
julia> function Base.show(io::IO, z::Polar)
           if get(io, :compact, false)::Bool
               print(io, z.r, "□", z.θ, "im")
           else
               print(io, z.r, " * exp(", z.θ, "im)")
           end
       end
```

This new compact representation will be used when the passed IO stream is an `IOContext` object with the `:compact` property set. In particular, this is the case when printing arrays with multiple columns (where horizontal space is limited):

```
julia> show(IOContext(stdout, :compact=>true), Polar(3, 4.0))
3.0□4.0im

julia> [Polar(3, 4.0) Polar(4.0,5.3)]
1×2 Matrix{Polar{Float64}}:
 3.0□4.0im  4.0□5.3im
```

See the `IOContext` documentation for a list of common properties which can be used to adjust printing.

Output-function summary

Here is a brief summary of the different output functions in Julia and how they are related. Most new types should only need to define `show` methods, if anything.

- `display(x)` tells the current environment to display `x` in whatever way it thinks best. (This might even be a graphical display in something like a Jupyter or Pluto notebook.) By default (e.g. in scripts or in the text REPL), it calls `show(io, "text/plain", x)`, or equivalently `show(io, MIME"text/plain"(), x)`, for an appropriate `io` stream. (In the REPL, `io` is an `IOContext` wrapper around `stdout`.) The REPL uses `display` to output the result of an evaluated expression.
- The 3-argument `show(io, ::MIME"text/plain", x)` method performs verbose pretty-printing of `x`. By default (if no 3-argument method is defined for `typeof(x)`), it calls the 2-argument `show(io, x)`. It is called by the 2-argument `repr("text/plain", x)`. Other 3-argument `show` methods can be defined for additional MIME types as discussed above, to enable richer display of `x` in some interactive environments.
- The 2-argument `show(io, x)` is the default simple text representation of `x`. It is called by the 1-argument `repr(x)`, and is typically the format you might employ to input `x` into Julia. The 1-argument `show(x)` calls `show(stdout, x)`.
- `print(io, x)` by default calls `show(io, x)`, but a few types have a distinct `print` format — most notably, when `x` is a string, `print` outputs the raw text whereas `show` outputs an escaped string enclosed in quotation marks. The 1-argument `print(x)` calls `print(stdout, x)`. `print` is also

called by `string(x)`. See also `println` (to append a newline) and `printstyled` (to add colors etc.), both of which call `print`.

- `write(io, x)`, if it is defined (it generally has *no* default definition for new types), writes a "raw" binary representation of x to `io`, e.g. an `x::Int32` will be written as 4 bytes.

It is also helpful to be familiar with the metadata that can be attached to an `io` stream by an `IOContext` wrapper. For example, the REPL sets the `:limit => true` flag from `display` for an evaluated expression, in order to limit the output to fit in the terminal; you can query this flag with `get(io, :limit, false)`. And when displaying an object contained within, for example, a multi-column matrix, the `:compact => true` flag could be set, which you can query with `get(io, :compact, false)`.

12.16 "Value types"

In Julia, you can't dispatch on a *value* such as `true` or `false`. However, you can dispatch on parametric types, and Julia allows you to include "plain bits" values (Types, Symbols, Integers, floating-point numbers, tuples, etc.) as type parameters. A common example is the dimensionality parameter in `Array{T,N}`, where `T` is a type (e.g., `Float64`) but `N` is just an `Int`.

You can create your own custom types that take values as parameters, and use them to control dispatch of custom types. By way of illustration of this idea, let's introduce the parametric type `Val{x}`, and its constructor `Val(x) = Val{x}()`, which serves as a customary way to exploit this technique for cases where you don't need a more elaborate hierarchy.

`Val` is defined as:

```
julia> struct Val{x}
       end

julia> Val(x) = Val{x}()
Val
```

There is no more to the implementation of `Val` than this. Some functions in Julia's standard library accept `Val` instances as arguments, and you can also use it to write your own functions. For example:

```
julia> firstlast(::Val{true}) = "First"
firstlast (generic function with 1 method)

julia> firstlast(::Val{false}) = "Last"
firstlast (generic function with 2 methods)

julia> firstlast(Val(true))
"First"

julia> firstlast(Val(false))
"Last"
```

For consistency across Julia, the call site should always pass a `Val` *instance* rather than using a *type*, i.e., use `foo(Val(:bar))` rather than `foo(Val{:bar})`.

It's worth noting that it's extremely easy to mis-use parametric "value" types, including `Val`; in unfavorable cases, you can easily end up making the performance of your code much *worse*. In particular, you would never want to write actual code as illustrated above. For more information about the proper (and improper) uses of `Val`, please read the more extensive discussion in the performance tips.

[1] "Small" is defined by the `max_union_splitting` configuration, which currently defaults to 4.

[2] A few popular languages have singleton types, including Haskell, Scala and Ruby.

Chapter 13

Methods

Recall from Functions that a function is an object that maps a tuple of arguments to a return value, or throws an exception if no appropriate value can be returned. It is common for the same conceptual function or operation to be implemented quite differently for different types of arguments: adding two integers is very different from adding two floating-point numbers, both of which are distinct from adding an integer to a floating-point number. Despite their implementation differences, these operations all fall under the general concept of "addition". Accordingly, in Julia, these behaviors all belong to a single object: the + function.

To facilitate using many different implementations of the same concept smoothly, functions need not be defined all at once, but can rather be defined piecewise by providing specific behaviors for certain combinations of argument types and counts. A definition of one possible behavior for a function is called a *method*. Thus far, we have presented only examples of functions defined with a single method, applicable to all types of arguments. However, the signatures of method definitions can be annotated to indicate the types of arguments in addition to their number, and more than a single method definition may be provided. When a function is applied to a particular tuple of arguments, the most specific method applicable to those arguments is applied. Thus, the overall behavior of a function is a patchwork of the behaviors of its various method definitions. If the patchwork is well designed, even though the implementations of the methods may be quite different, the outward behavior of the function will appear seamless and consistent.

The choice of which method to execute when a function is applied is called *dispatch*. Julia allows the dispatch process to choose which of a function's methods to call based on the number of arguments given, and on the types of all of the function's arguments. This is different than traditional object-oriented languages, where dispatch occurs based only on the first argument, which often has a special argument syntax, and is sometimes implied rather than explicitly written as an argument. [1] Using all of a function's arguments to choose which method should be invoked, rather than just the first, is known as multiple dispatch. Multiple dispatch is particularly useful for mathematical code, where it makes little sense to artificially deem the operations to "belong" to one argument more than any of the others: does the addition operation in `x + y` belong to x any more than it does to y? The implementation of a mathematical operator generally depends on the types of all of its arguments. Even beyond mathematical operations, however, multiple dispatch ends up being a powerful and convenient paradigm for structuring and organizing programs.

[1] In C++ or Java, for example, in a method call like `obj.meth(arg1,arg2)`, the object obj "receives" the method call and is implicitly passed to the method via the `this` keyword, rather than as an explicit method argument. When the current `this` object is the receiver of a method call, it can be omitted altogether, writing just `meth(arg1,arg2)`, with `this` implied as the receiving object.

Note

All the examples in this chapter assume that you are defining methods for a function in the *same* module. If you want to add methods to a function in *another* module, you have to `import` it or use the name qualified with module names. See the section on namespace management.

13.1 Defining Methods

Until now, we have, in our examples, defined only functions with a single method having unconstrained argument types. Such functions behave just like they would in traditional dynamically typed languages. Nevertheless, we have used multiple dispatch and methods almost continually without being aware of it: all of Julia's standard functions and operators, like the aforementioned + function, have many methods defining their behavior over various possible combinations of argument type and count.

When defining a function, one can optionally constrain the types of parameters it is applicable to, using the `::` type-assertion operator, introduced in the section on Composite Types:

```
julia> f(x::Float64, y::Float64) = 2x + y
f (generic function with 1 method)
```

This function definition applies only to calls where x and y are both values of type `Float64`:

```
julia> f(2.0, 3.0)
7.0
```

Applying it to any other types of arguments will result in a `MethodError`:

```
julia> f(2.0, 3)
ERROR: MethodError: no method matching f(::Float64, ::Int64)
The function `f` exists, but no method is defined for this combination of argument types.

Closest candidates are:
  f(::Float64, !Matched::Float64)
   @ Main none:1

Stacktrace:
[...]

julia> f(Float32(2.0), 3.0)
ERROR: MethodError: no method matching f(::Float32, ::Float64)
The function `f` exists, but no method is defined for this combination of argument types.

Closest candidates are:
  f(!Matched::Float64, ::Float64)
   @ Main none:1

Stacktrace:
[...]

julia> f(2.0, "3.0")
```

```
ERROR: MethodError: no method matching f(::Float64, ::String)
The function `f` exists, but no method is defined for this combination of argument types.

Closest candidates are:
  f(::Float64, !Matched::Float64)
   @ Main none:1

Stacktrace:
[...]

julia> f("2.0", "3.0")
ERROR: MethodError: no method matching f(::String, ::String)
The function `f` exists, but no method is defined for this combination of argument types.
```

As you can see, the arguments must be precisely of type `Float64`. Other numeric types, such as integers or 32-bit floating-point values, are not automatically converted to 64-bit floating-point, nor are strings parsed as numbers. Because `Float64` is a concrete type and concrete types cannot be subclassed in Julia, such a definition can only be applied to arguments that are exactly of type `Float64`. It may often be useful, however, to write more general methods where the declared parameter types are abstract:

```
julia> f(x::Number, y::Number) = 2x - y
f (generic function with 2 methods)

julia> f(2.0, 3)
1.0
```

This method definition applies to any pair of arguments that are instances of `Number`. They need not be of the same type, so long as they are each numeric values. The problem of handling disparate numeric types is delegated to the arithmetic operations in the expression `2x - y`.

To define a function with multiple methods, one simply defines the function multiple times, with different numbers and types of arguments. The first method definition for a function creates the function object, and subsequent method definitions add new methods to the existing function object. The most specific method definition matching the number and types of the arguments will be executed when the function is applied. Thus, the two method definitions above, taken together, define the behavior for `f` over all pairs of instances of the abstract type `Number` – but with a different behavior specific to pairs of `Float64` values. If one of the arguments is a 64-bit float but the other one is not, then the `f(Float64,Float64)` method cannot be called and the more general `f(Number,Number)` method must be used:

```
julia> f(2.0, 3.0)
7.0

julia> f(2, 3.0)
1.0

julia> f(2.0, 3)
1.0

julia> f(2, 3)
1
```

The 2x + y definition is only used in the first case, while the 2x - y definition is used in the others. No automatic casting or conversion of function arguments is ever performed: all conversion in Julia is non-magical and completely explicit. Conversion and Promotion, however, shows how clever application of sufficiently advanced technology can be indistinguishable from magic. [2]

For non-numeric values, and for fewer or more than two arguments, the function f remains undefined, and applying it will still result in a MethodError:

```
julia> f("foo", 3)
ERROR: MethodError: no method matching f(::String, ::Int64)
The function `f` exists, but no method is defined for this combination of argument types.

Closest candidates are:
  f(!Matched::Number, ::Number)
   @ Main none:1
  f(!Matched::Float64, !Matched::Float64)
   @ Main none:1

Stacktrace:
[...]

julia> f()
ERROR: MethodError: no method matching f()
The function `f` exists, but no method is defined for this combination of argument types.

Closest candidates are:
  f(!Matched::Float64, !Matched::Float64)
   @ Main none:1
  f(!Matched::Number, !Matched::Number)
   @ Main none:1

Stacktrace:
[...]
```

You can easily see which methods exist for a function by entering the function object itself in an interactive session:

```
julia> f
f (generic function with 2 methods)
```

This output tells us that f is a function object with two methods. To find out what the signatures of those methods are, use the methods function:

```
julia> methods(f)
# 2 methods for generic function "f" from Main:
 [1] f(x::Float64, y::Float64)
     @ none:1
 [2] f(x::Number, y::Number)
     @ none:1
```

which shows that `f` has two methods, one taking two `Float64` arguments and one taking arguments of type `Number`. It also indicates the file and line number where the methods were defined: because these methods were defined at the REPL, we get the apparent line number `none:1`.

In the absence of a type declaration with `::`, the type of a method parameter is `Any` by default, meaning that it is unconstrained since all values in Julia are instances of the abstract type `Any`. Thus, we can define a catch-all method for `f` like so:

```
julia> f(x,y) = println("Whoa there, Nelly.")
f (generic function with 3 methods)

julia> methods(f)
# 3 methods for generic function "f" from Main:
 [1] f(x::Float64, y::Float64)
     @ none:1
 [2] f(x::Number, y::Number)
     @ none:1
 [3] f(x, y)
     @ none:1

julia> f("foo", 1)
Whoa there, Nelly.
```

This catch-all is less specific than any other possible method definition for a pair of parameter values, so it will only be called on pairs of arguments to which no other method definition applies.

Note that in the signature of the third method, there is no type specified for the arguments `x` and `y`. This is a shortened way of expressing `f(x::Any, y::Any)`.

Although it seems a simple concept, multiple dispatch on the types of values is perhaps the single most powerful and central feature of the Julia language. Core operations typically have dozens of methods:

```
julia> methods(+)
# 180 methods for generic function "+":
[1] +(x::Bool, z::Complex{Bool}) in Base at complex.jl:227
[2] +(x::Bool, y::Bool) in Base at bool.jl:89
[3] +(x::Bool) in Base at bool.jl:86
[4] +(x::Bool, y::T) where T<:AbstractFloat in Base at bool.jl:96
[5] +(x::Bool, z::Complex) in Base at complex.jl:234
[6] +(a::Float16, b::Float16) in Base at float.jl:373
[7] +(x::Float32, y::Float32) in Base at float.jl:375
[8] +(x::Float64, y::Float64) in Base at float.jl:376
[9] +(z::Complex{Bool}, x::Bool) in Base at complex.jl:228
[10] +(z::Complex{Bool}, x::Real) in Base at complex.jl:242
[11] +(x::Char, y::Integer) in Base at char.jl:40
[12] +(c::BigInt, x::BigFloat) in Base.MPFR at mpfr.jl:307
[13] +(a::BigInt, b::BigInt, c::BigInt, d::BigInt, e::BigInt) in Base.GMP at gmp.jl:392
[14] +(a::BigInt, b::BigInt, c::BigInt, d::BigInt) in Base.GMP at gmp.jl:391
[15] +(a::BigInt, b::BigInt, c::BigInt) in Base.GMP at gmp.jl:390
[16] +(x::BigInt, y::BigInt) in Base.GMP at gmp.jl:361
[17] +(x::BigInt, c::Union{UInt16, UInt32, UInt64, UInt8}) in Base.GMP at gmp.jl:398
...
[180] +(a, b, c, xs...) in Base at operators.jl:424
```

Multiple dispatch together with the flexible parametric type system give Julia its ability to abstractly express high-level algorithms decoupled from implementation details.

13.2 Method specializations

When you create multiple methods of the same function, this is sometimes called "specialization." In this case, you're specializing the *function* by adding additional methods to it: each new method is a new specialization of the function. As shown above, these specializations are returned by `methods`.

There's another kind of specialization that occurs without programmer intervention: Julia's compiler can automatically specialize the *method* for the specific argument types used. Such specializations are *not* listed by `methods`, as this doesn't create new `Methods`, but tools like `@code_typed` allow you to inspect such specializations.

For example, if you create a method

```
mysum(x::Real, y::Real) = x + y
```

you've given the function `mysum` one new method (possibly its only method), and that method takes any pair of `Real` number inputs. But if you then execute

```
julia> mysum(1, 2)
3

julia> mysum(1.0, 2.0)
3.0
```

Julia will compile `mysum` twice, once for `x::Int, y::Int` and again for `x::Float64, y::Float64`. The point of compiling twice is performance: the methods that get called for `+` (which `mysum` uses) vary depending on the specific types of `x` and `y`, and by compiling different specializations Julia can do all the method lookup ahead of time. This allows the program to run much more quickly, since it does not have to bother with method lookup while it is running. Julia's automatic specialization allows you to write generic algorithms and expect that the compiler will generate efficient, specialized code to handle each case you need.

In cases where the number of potential specializations might be effectively unlimited, Julia may avoid this default specialization. See Be aware of when Julia avoids specializing for more information.

13.3 Method Ambiguities

It is possible to define a set of function methods such that there is no unique most specific method applicable to some combinations of arguments:

```
julia> g(x::Float64, y) = 2x + y
g (generic function with 1 method)

julia> g(x, y::Float64) = x + 2y
g (generic function with 2 methods)

julia> g(2.0, 3)
7.0
```

```
julia> g(2, 3.0)
8.0

julia> g(2.0, 3.0)
ERROR: MethodError: g(::Float64, ::Float64) is ambiguous.

Candidates:
  g(x, y::Float64)
    @ Main none:1
  g(x::Float64, y)
    @ Main none:1

Possible fix, define
  g(::Float64, ::Float64)

Stacktrace:
[...]
```

Here the call `g(2.0, 3.0)` could be handled by either the `g(::Float64, ::Any)` or the `g(::Any, ::Float64)` method. The order in which the methods are defined does not matter and neither is more specific than the other. In such cases, Julia raises a `MethodError` rather than arbitrarily picking a method. You can avoid method ambiguities by specifying an appropriate method for the intersection case:

```
julia> g(x::Float64, y::Float64) = 2x + 2y
g (generic function with 3 methods)

julia> g(2.0, 3)
7.0

julia> g(2, 3.0)
8.0

julia> g(2.0, 3.0)
10.0
```

It is recommended that the disambiguating method be defined first, since otherwise the ambiguity exists, if transiently, until the more specific method is defined.

In more complex cases, resolving method ambiguities involves a certain element of design; this topic is explored further below.

13.4 Parametric Methods

Method definitions can optionally have type parameters qualifying the signature:

```
julia> same_type(x::T, y::T) where {T} = true
same_type (generic function with 1 method)

julia> same_type(x,y) = false
same_type (generic function with 2 methods)
```

The first method applies whenever both arguments are of the same concrete type, regardless of what type that is, while the second method acts as a catch-all, covering all other cases. Thus, overall, this defines a boolean function that checks whether its two arguments are of the same type:

```
julia> same_type(1, 2)
true

julia> same_type(1, 2.0)
false

julia> same_type(1.0, 2.0)
true

julia> same_type("foo", 2.0)
false

julia> same_type("foo", "bar")
true

julia> same_type(Int32(1), Int64(2))
false
```

Such definitions correspond to methods whose type signatures are `UnionAll` types (see UnionAll Types).

This kind of definition of function behavior by dispatch is quite common – idiomatic, even – in Julia. Method type parameters are not restricted to being used as the types of arguments: they can be used anywhere a value would be in the signature of the function or body of the function. Here's an example where the method type parameter T is used as the type parameter to the parametric type `Vector{T}` in the method signature:

```
julia> function myappend(v::Vector{T}, x::T) where {T}
           return [v..., x]
       end
myappend (generic function with 1 method)
```

The type parameter T in this example ensures that the added element x is a subtype of the existing eltype of the vector v. The `where` keyword introduces a list of those constraints after the method signature definition. This works the same for one-line definitions, as seen above, and must appear *before* the return type declaration, if present, as illustrated below:

```
julia> (myappend(v::Vector{T}, x::T)::Vector) where {T} = [v..., x]
myappend (generic function with 1 method)

julia> myappend([1,2,3],4)
4-element Vector{Int64}:
 1
 2
 3
 4

julia> myappend([1,2,3],2.5)
```

```
ERROR: MethodError: no method matching myappend(::Vector{Int64}, ::Float64)
The function `myappend` exists, but no method is defined for this combination of argument types.

Closest candidates are:
  myappend(::Vector{T}, !Matched::T) where T
   @ Main none:1

Stacktrace:
[...]

julia> myappend([1.0,2.0,3.0],4.0)
4-element Vector{Float64}:
 1.0
 2.0
 3.0
 4.0

julia> myappend([1.0,2.0,3.0],4)
ERROR: MethodError: no method matching myappend(::Vector{Float64}, ::Int64)
The function `myappend` exists, but no method is defined for this combination of argument types.

Closest candidates are:
  myappend(::Vector{T}, !Matched::T) where T
   @ Main none:1

Stacktrace:
[...]
```

If the type of the appended element does not match the element type of the vector it is appended to, a `MethodError` is raised. In the following example, the method's type parameter T is used as the return value:

```
julia> mytypeof(x::T) where {T} = T
mytypeof (generic function with 1 method)

julia> mytypeof(1)
Int64

julia> mytypeof(1.0)
Float64
```

Just as you can put subtype constraints on type parameters in type declarations (see Parametric Types), you can also constrain type parameters of methods:

```
julia> same_type_numeric(x::T, y::T) where {T<:Number} = true
same_type_numeric (generic function with 1 method)

julia> same_type_numeric(x::Number, y::Number) = false
same_type_numeric (generic function with 2 methods)

julia> same_type_numeric(1, 2)
```

```
true

julia> same_type_numeric(1, 2.0)
false

julia> same_type_numeric(1.0, 2.0)
true

julia> same_type_numeric("foo", 2.0)
ERROR: MethodError: no method matching same_type_numeric(::String, ::Float64)
The function `same_type_numeric` exists, but no method is defined for this combination of
↪  argument types.

Closest candidates are:
  same_type_numeric(!Matched::T, ::T) where T<:Number
   @ Main none:1
  same_type_numeric(!Matched::Number, ::Number)
   @ Main none:1

Stacktrace:
[...]

julia> same_type_numeric("foo", "bar")
ERROR: MethodError: no method matching same_type_numeric(::String, ::String)
The function `same_type_numeric` exists, but no method is defined for this combination of
↪  argument types.

julia> same_type_numeric(Int32(1), Int64(2))
false
```

The `same_type_numeric` function behaves much like the `same_type` function defined above, but is only defined for pairs of numbers.

Parametric methods allow the same syntax as `where` expressions used to write types (see UnionAll Types). If there is only a single parameter, the enclosing curly braces (in `where {T}`) can be omitted, but are often preferred for clarity. Multiple parameters can be separated with commas, e.g. `where {T, S<:Real}`, or written using nested `where`, e.g. `where S<:Real where T`.

13.5 Redefining Methods

When redefining a method or adding new methods, it is important to realize that these changes don't take effect immediately. This is key to Julia's ability to statically infer and compile code to run fast, without the usual JIT tricks and overhead. Indeed, any new method definition won't be visible to the current runtime environment, including Tasks and Threads (and any previously defined `@generated` functions). Let's start with an example to see what this means:

```
julia> function tryeval()
           @eval newfun() = 1
           newfun()
       end
tryeval (generic function with 1 method)
```

```
julia> tryeval()
ERROR: MethodError: no method matching newfun()
The applicable method may be too new: running in world age xxxx1, while current world is xxxx2.
Closest candidates are:
  newfun() at none:1 (method too new to be called from this world context.)
 in tryeval() at none:1
 ...

julia> newfun()
1
```

In this example, observe that the new definition for `newfun` has been created, but can't be immediately called. The new global is immediately visible to the `tryeval` function, so you could write `return newfun` (without parentheses). But neither you, nor any of your callers, nor the functions they call, or etc. can call this new method definition!

But there's an exception: future calls to `newfun` *from the REPL* work as expected, being able to both see and call the new definition of `newfun`.

However, future calls to `tryeval` will continue to see the definition of `newfun` as it was *at the previous statement at the REPL*, and thus before that call to `tryeval`.

You may want to try this for yourself to see how it works.

The implementation of this behavior is a "world age counter", which is further described in the World Age manual chapter.

13.6 Design Patterns with Parametric Methods

While complex dispatch logic is not required for performance or usability, sometimes it can be the best way to express some algorithm. Here are a few common design patterns that come up sometimes when using dispatch in this way.

Extracting the type parameter from a super-type

Here is a correct code template for returning the element-type `T` of any arbitrary subtype of `AbstractArray` that has well-defined element type:

```
abstract type AbstractArray{T, N} end
eltype(::Type{<:AbstractArray{T}}) where {T} = T
```

using so-called triangular dispatch. Note that `UnionAll` types, for example `eltype(AbstractArray{T} where T <: Integer)`, do not match the above method. The implementation of `eltype` in `Base` adds a fallback method to `Any` for such cases.

One common mistake is to try and get the element-type by using introspection:

```
eltype_wrong(::Type{A}) where {A<:AbstractArray} = A.parameters[1]
```

However, it is not hard to construct cases where this will fail:

```
struct BitVector <: AbstractArray{Bool, 1}; end
```

Here we have created a type `BitVector` which has no parameters, but where the element-type is still fully specified, with T equal to `Bool`!

Another mistake is to try to walk up the type hierarchy using `supertype`:

```
eltype_wrong(::Type{AbstractArray{T}}) where {T} = T
eltype_wrong(::Type{AbstractArray{T, N}}) where {T, N} = T
eltype_wrong(::Type{A}) where {A<:AbstractArray} = eltype_wrong(supertype(A))
```

While this works for declared types, it fails for types without supertypes:

```
julia> eltype_wrong(Union{Vector{Int}, Matrix{Int}})
ERROR: MethodError: no method matching supertype(::Type{VecOrMat{Int64}})

Closest candidates are:
  supertype(::UnionAll)
   @ Base operators.jl:44
  supertype(::DataType)
   @ Base operators.jl:43
```

Building a similar type with a different type parameter

When building generic code, there is often a need for constructing a similar object with some change made to the layout of the type, also necessitating a change of the type parameters. For instance, you might have some sort of abstract array with an arbitrary element type and want to write your computation on it with a specific element type. We must implement a method for each `AbstractArray{T}` subtype that describes how to compute this type transform. There is no general transform of one subtype into another subtype with a different parameter.

The subtypes of `AbstractArray` typically implement two methods to achieve this: A method to convert the input array to a subtype of a specific `AbstractArray{T, N}` abstract type; and a method to make a new uninitialized array with a specific element type. Sample implementations of these can be found in Julia Base. Here is a basic example usage of them, guaranteeing that `input` and `output` are of the same type:

```
input = convert(AbstractArray{Eltype}, input)
output = similar(input, Eltype)
```

As an extension of this, in cases where the algorithm needs a copy of the input array, `convert` is insufficient as the return value may alias the original input. Combining `similar` (to make the output array) and `copyto!` (to fill it with the input data) is a generic way to express the requirement for a mutable copy of the input argument:

```
copy_with_eltype(input, Eltype) = copyto!(similar(input, Eltype), input)
```

Iterated dispatch

In order to dispatch a multi-level parametric argument list, often it is best to separate each level of dispatch into distinct functions. This may sound similar in approach to single-dispatch, but as we shall see below, it is still more flexible.

For example, trying to dispatch on the element-type of an array will often run into ambiguous situations. Instead, common code will dispatch first on the container type, then recurse down to a more specific method based on `eltype`. In most cases, the algorithms lend themselves conveniently to this hierarchical approach, while in other cases, this rigor must be resolved manually. This dispatching branching can be observed, for example, in the logic to sum two matrices:

```
# First dispatch selects the map algorithm for element-wise summation.
+(a::Matrix, b::Matrix) = map(+, a, b)
# Then dispatch handles each element and selects the appropriate
# common element type for the computation.
+(a, b) = +(promote(a, b)...)
# Once the elements have the same type, they can be added.
# For example, via primitive operations exposed by the processor.
+(a::Float64, b::Float64) = Core.add(a, b)
```

Trait-based dispatch

A natural extension to the iterated dispatch above is to add a layer to method selection that allows to dispatch on sets of types which are independent from the sets defined by the type hierarchy. We could construct such a set by writing out a `Union` of the types in question, but then this set would not be extensible as `Union`-types cannot be altered after creation. However, such an extensible set can be programmed with a design pattern often referred to as a "Holy-trait".

This pattern is implemented by defining a generic function which computes a different singleton value (or type) for each trait-set to which the function arguments may belong to. If this function is pure there is no impact on performance compared to normal dispatch.

The example in the previous section glossed over the implementation details of `map` and `promote`, which both operate in terms of these traits. When iterating over a matrix, such as in the implementation of `map`, one important question is what order to use to traverse the data. When `AbstractArray` subtypes implement the `Base.IndexStyle` trait, other functions such as `map` can dispatch on this information to pick the best algorithm (see Abstract Array Interface). This means that each subtype does not need to implement a custom version of `map`, since the generic definitions + trait classes will enable the system to select the fastest version. Here is a toy implementation of `map` illustrating the trait-based dispatch:

```
map(f, a::AbstractArray, b::AbstractArray) = map(Base.IndexStyle(a, b), f, a, b)
# generic implementation:
map(::Base.IndexCartesian, f, a::AbstractArray, b::AbstractArray) = ...
# linear-indexing implementation (faster)
map(::Base.IndexLinear, f, a::AbstractArray, b::AbstractArray) = ...
```

This trait-based approach is also present in the `promote` mechanism employed by the scalar `+`. It uses `promote_type`, which returns the optimal common type to compute the operation given the two types of the operands. This makes it possible to reduce the problem of implementing every function for every pair of possible type arguments, to the much smaller problem of implementing a conversion operation from each type to a common type, plus a table of preferred pair-wise promotion rules.

Output-type computation

The discussion of trait-based promotion provides a transition into our next design pattern: computing the output element type for a matrix operation.

For implementing primitive operations, such as addition, we use the `promote_type` function to compute the desired output type. (As before, we saw this at work in the `promote` call in the call to +).

For more complex functions on matrices, it may be necessary to compute the expected return type for a more complex sequence of operations. This is often performed by the following steps:

1. Write a small function `op` that expresses the set of operations performed by the kernel of the algorithm.
2. Compute the element type R of the result matrix as `promote_op(op, argument_types...)`, where `argument_types` is computed from `eltype` applied to each input array.
3. Build the output matrix as `similar(R, dims)`, where `dims` are the desired dimensions of the output array.

For a more specific example, a generic square-matrix multiply pseudo-code might look like:

```
function matmul(a::AbstractMatrix, b::AbstractMatrix)
    op = (ai, bi) -> ai * bi + ai * bi

    ## this is insufficient because it assumes `one(eltype(a))` is constructable:
    # R = typeof(op(one(eltype(a)), one(eltype(b))))

    ## this fails because it assumes `a[1]` exists and is representative of all elements of the
    ↪  array
    # R = typeof(op(a[1], b[1]))

    ## this is incorrect because it assumes that `+` calls `promote_type`
    ## but this is not true for some types, such as Bool:
    # R = promote_type(ai, bi)

    # this is wrong, since depending on the return value
    # of type-inference is very brittle (as well as not being optimizable):
    # R = Base.return_types(op, (eltype(a), eltype(b)))

    ## but, finally, this works:
    R = promote_op(op, eltype(a), eltype(b))
    ## although sometimes it may give a larger type than desired
    ## it will always give a correct type

    output = similar(b, R, (size(a, 1), size(b, 2)))
    if size(a, 2) > 0
        for j in 1:size(b, 2)
            for i in 1:size(a, 1)
                ## here we don't use `ab = zero(R)`,
                ## since `R` might be `Any` and `zero(Any)` is not defined
                ## we also must declare `ab::R` to make the type of `ab` constant in the loop,
                ## since it is possible that typeof(a * b) != typeof(a * b + a * b) == R
```

```
                ab::R = a[i, 1] * b[1, j]
                for k in 2:size(a, 2)
                    ab += a[i, k] * b[k, j]
                end
                output[i, j] = ab
            end
        end
    end
    return output
end
```

Separate convert and kernel logic

One way to significantly cut down on compile-times and testing complexity is to isolate the logic for converting to the desired type and the computation. This lets the compiler specialize and inline the conversion logic independent from the rest of the body of the larger kernel.

This is a common pattern seen when converting from a larger class of types to the one specific argument type that is actually supported by the algorithm:

```
complexfunction(arg::Int) = ...
complexfunction(arg::Any) = complexfunction(convert(Int, arg))

matmul(a::T, b::T) = ...
matmul(a, b) = matmul(promote(a, b)...)
```

13.7 Parametrically-constrained Varargs methods

Function parameters can also be used to constrain the number of arguments that may be supplied to a "varargs" function (Varargs Functions). The notation `Vararg{T,N}` is used to indicate such a constraint. For example:

```
julia> bar(a,b,x::Vararg{Any,2}) = (a,b,x)
bar (generic function with 1 method)

julia> bar(1,2,3)
ERROR: MethodError: no method matching bar(::Int64, ::Int64, ::Int64)
The function `bar` exists, but no method is defined for this combination of argument types.

Closest candidates are:
  bar(::Any, ::Any, ::Any, !Matched::Any)
   @ Main none:1

Stacktrace:
[...]

julia> bar(1,2,3,4)
(1, 2, (3, 4))

julia> bar(1,2,3,4,5)
ERROR: MethodError: no method matching bar(::Int64, ::Int64, ::Int64, ::Int64, ::Int64)
The function `bar` exists, but no method is defined for this combination of argument types.
```

```
Closest candidates are:
  bar(::Any, ::Any, ::Any, ::Any)
   @ Main none:1

Stacktrace:
[...]
```

More usefully, it is possible to constrain varargs methods by a parameter. For example:

```
function getindex(A::AbstractArray{T,N}, indices::Vararg{Number,N}) where {T,N}
```

would be called only when the number of `indices` matches the dimensionality of the array.

When only the type of supplied arguments needs to be constrained `Vararg{T}` can be equivalently written as `T...`. For instance `f(x::Int...) = x` is a shorthand for `f(x::Vararg{Int}) = x`.

13.8 Note on Optional and keyword Arguments

As mentioned briefly in Functions, optional arguments are implemented as syntax for multiple method definitions. For example, this definition:

```
f(a=1,b=2) = a+2b
```

translates to the following three methods:

```
f(a,b) = a+2b
f(a) = f(a,2)
f() = f(1,2)
```

This means that calling `f()` is equivalent to calling `f(1,2)`. In this case the result is 5, because `f(1,2)` invokes the first method of `f` above. However, this need not always be the case. If you define a fourth method that is more specialized for integers:

```
f(a::Int,b::Int) = a-2b
```

then the result of both `f()` and `f(1,2)` is -3. In other words, optional arguments are tied to a function, not to any specific method of that function. It depends on the types of the optional arguments which method is invoked. When optional arguments are defined in terms of a global variable, the type of the optional argument may even change at run-time.

Keyword arguments behave quite differently from ordinary positional arguments. In particular, they do not participate in method dispatch. Methods are dispatched based only on positional arguments, with keyword arguments processed after the matching method is identified.

Known bug: the presence of keyword arguments affects dispatch

Due to a long-standing bug (#9498), a call that supplies keyword arguments only considers methods that accept keyword arguments. This can select a less specific method over a more specific one that accepts no keywords:

```
julia> f(x; y=10) = "generic";

julia> f(x::Int) = "int-specific";

julia> f(1)
"int-specific"

julia> f(1; y=2)  # bug: bypasses the more specific method
"generic"
```

Relatedly, redefining a method without keyword arguments does not replace the keyword handling of an earlier definition with the same positional signature, even though only one method is displayed:

```
julia> g(x; y=1) = "with keywords";

julia> g(x) = "without keywords";

julia> g
g (generic function with 1 method)

julia> g(1)
"without keywords"

julia> g(1; y=2)  # bug: calls the overwritten definition
"with keywords"
```

This warning is descriptive, not prescriptive, and code should not rely on this behavior. To keep a specialized method applicable to calls with keyword arguments, give it its own keyword interface, either by repeating the keyword arguments or by collecting any keyword with `kwargs...`.

13.9 Function-like objects

Methods are associated with types, so it is possible to make any arbitrary Julia object "callable" by adding methods to its type.

For example, you can define a type that stores the coefficients of a polynomial, but behaves like a function evaluating the polynomial:

```
julia> struct Polynomial{R}
           coeffs::Vector{R}
       end
```

```
julia> function (p::Polynomial)(x)
           v = p.coeffs[end]
           for i = (length(p.coeffs)-1):-1:1
               v = v*x + p.coeffs[i]
           end
           return v
       end

julia> (p::Polynomial)() = p(5)
```

Notice that the function is specified by type instead of by name. As with normal functions there is a terse syntax form. In the function body, `p` will refer to the object that was called. A `Polynomial` can be used as follows:

```
julia> poly = Polynomial([1,10,100])
Polynomial{Int64}([1, 10, 100])

julia> poly(3)
931

julia> poly()
2551
```

This mechanism is also the key to how type constructors and closures (inner functions that refer to their surrounding environment) work in Julia.

13.10 Empty generic functions

Occasionally it is useful to introduce a generic function without yet adding methods. This can be used to separate interface definitions from implementations. It might also be done for the purpose of documentation or code readability. The syntax for this is an empty `function` block without a tuple of arguments:

```
function emptyfunc end
```

13.11 Method design and the avoidance of ambiguities

Julia's method polymorphism is one of its most powerful features, yet exploiting this power can pose design challenges. In particular, in more complex method hierarchies it is not uncommon for ambiguities to arise.

Above, it was pointed out that one can resolve ambiguities like

```
f(x, y::Int) = 1
f(x::Int, y) = 2
```

by defining a method

```
f(x::Int, y::Int) = 3
```

This is often the right strategy; however, there are circumstances where following this advice mindlessly can be counterproductive. In particular, the more methods a generic function has, the more possibilities there are for ambiguities. When your method hierarchies get more complicated than this simple example, it can be worth your while to think carefully about alternative strategies.

Below we discuss particular challenges and some alternative ways to resolve such issues.

Tuple and NTuple arguments

`Tuple` (and `NTuple`) arguments present special challenges. For example,

```
f(x::NTuple{N,Int}) where {N} = 1
f(x::NTuple{N,Float64}) where {N} = 2
```

are ambiguous because of the possibility that `N == 0`: there are no elements to determine whether the `Int` or `Float64` variant should be called. To resolve the ambiguity, one approach is define a method for the empty tuple:

```
f(x::Tuple{}) = 3
```

Alternatively, for all methods but one you can insist that there is at least one element in the tuple:

```
f(x::NTuple{N,Int}) where {N} = 1           # this is the fallback
f(x::Tuple{Float64, Vararg{Float64}}) = 2   # this requires at least one Float64
```

Orthogonalize your design

When you might be tempted to dispatch on two or more arguments, consider whether a "wrapper" function might make for a simpler design. For example, instead of writing multiple variants:

```
f(x::A, y::A) = ...
f(x::A, y::B) = ...
f(x::B, y::A) = ...
f(x::B, y::B) = ...
```

you might consider defining

```
f(x::A, y::A) = ...
f(x, y) = f(g(x), g(y))
```

where `g` converts the argument to type `A`. This is a very specific example of the more general principle of orthogonal design, in which separate concepts are assigned to separate methods. Here, `g` will most likely need a fallback definition

```
g(x::A) = x
```

A related strategy exploits `promote` to bring `x` and `y` to a common type:

```
f(x::T, y::T) where {T} = ...
f(x, y) = f(promote(x, y)...)
```

One risk with this design is the possibility that if there is no suitable promotion method converting x and y to the same type, the second method will recurse on itself infinitely and trigger a stack overflow.

Dispatch on one argument at a time

If you need to dispatch on multiple arguments, and there are many fallbacks with too many combinations to make it practical to define all possible variants, then consider introducing a "name cascade" where (for example) you dispatch on the first argument and then call an internal method:

```
f(x::A, y) = _fA(x, y)
f(x::B, y) = _fB(x, y)
```

Then the internal methods `_fA` and `_fB` can dispatch on y without concern about ambiguities with each other with respect to x.

Be aware that this strategy has at least one major disadvantage: in many cases, it is not possible for users to further customize the behavior of `f` by defining further specializations of your exported function `f`. Instead, they have to define specializations for your internal methods `_fA` and `_fB`, and this blurs the lines between exported and internal methods.

Abstract containers and element types

Where possible, try to avoid defining methods that dispatch on specific element types of abstract containers. For example,

```
-(A::AbstractArray{T}, b::Date) where {T<:Date}
```

generates ambiguities for anyone who defines a method

```
-(A::MyArrayType{T}, b::T) where {T}
```

The best approach is to avoid defining *either* of these methods: instead, rely on a generic method `-(A::AbstractArray, b)` and make sure this method is implemented with generic calls (like `similar` and `-`) that do the right thing for each container type and element type *separately*. This is just a more complex variant of the advice to orthogonalize your methods.

When this approach is not possible, it may be worth starting a discussion with other developers about resolving the ambiguity; just because one method was defined first does not necessarily mean that it can't be modified or eliminated. As a last resort, one developer can define the "band-aid" method

```
-(A::MyArrayType{T}, b::Date) where {T<:Date} = ...
```

that resolves the ambiguity by brute force.

Complex method "cascades" with default arguments

If you are defining a method "cascade" that supplies defaults, be careful about dropping any arguments that correspond to potential defaults. For example, suppose you're writing a digital filtering algorithm and you have a method that handles the edges of the signal by applying padding:

```
function myfilter(A, kernel, ::Replicate)
    Apadded = replicate_edges(A, size(kernel))
    myfilter(Apadded, kernel)  # now perform the "real" computation
end
```

This will run afoul of a method that supplies default padding:

```
myfilter(A, kernel) = myfilter(A, kernel, Replicate()) # replicate the edge by default
```

Together, these two methods generate an infinite recursion with A constantly growing bigger.

The better design would be to define your call hierarchy like this:

```
struct NoPad end  # indicate that no padding is desired, or that it's already applied

myfilter(A, kernel) = myfilter(A, kernel, Replicate())  # default boundary conditions

function myfilter(A, kernel, ::Replicate)
    Apadded = replicate_edges(A, size(kernel))
    myfilter(Apadded, kernel, NoPad())  # indicate the new boundary conditions
end

# other padding methods go here

function myfilter(A, kernel, ::NoPad)
    # Here's the "real" implementation of the core computation
end
```

NoPad is supplied in the same argument position as any other kind of padding, so it keeps the dispatch hierarchy well organized and with reduced likelihood of ambiguities. Moreover, it extends the "public" `myfilter` interface: a user who wants to control the padding explicitly can call the NoPad variant directly.

13.12 Defining methods in local scope

You can define methods within a local scope, for example

```
julia> function f(x)
           g(y::Int) = y + x
           g(y) = y - x
           g
       end
f (generic function with 1 method)

julia> h = f(3);
```

```
julia> h(4)
7

julia> h(4.0)
1.0
```

However, you should *not* define local methods conditionally or subject to control flow, as in

```
function f2(inc)
    if inc
        g(x) = x + 1
    else
        g(x) = x - 1
    end
end

function f3()
    function g end
    return g
    g() = 0
end
```

as it is not clear what function will end up getting defined. In the future, it might be an error to define local methods in this manner.

For cases like this use anonymous functions instead:

```
function f2(inc)
    g = if inc
        x -> x + 1
    else
        x -> x - 1
    end
end
```

[2] Arthur C. Clarke, *Profiles of the Future* (1961): Clarke's Third Law.

Chapter 14

Constructors

Constructors [1] are functions that create new objects – specifically, instances of Composite Types. In Julia, type objects also serve as constructor functions: they create new instances of themselves when applied to an argument tuple as a function. This much was already mentioned briefly when composite types were introduced. For example:

```
julia> struct Foo
           bar
           baz
       end

julia> foo = Foo(1, 2)
Foo(1, 2)

julia> foo.bar
1

julia> foo.baz
2
```

For many types, forming new objects by binding their field values together is all that is ever needed to create instances. However, in some cases more functionality is required when creating composite objects. Sometimes invariants must be enforced, either by checking arguments or by transforming them. Recursive data structures, especially those that may be self-referential, often cannot be constructed cleanly without first being created in an incomplete state and then altered programmatically to be made whole, as a separate step from object creation. Sometimes, it's just convenient to be able to construct objects with fewer or different types of parameters than they have fields. Julia's system for object construction addresses all of these cases and more.

[1] Nomenclature: while the term "constructor" generally refers to the entire function which constructs objects of a type, it is common to abuse terminology slightly and refer to specific constructor methods as "constructors". In such situations, it is generally clear from the context that the term is used to mean "constructor method" rather than "constructor function", especially as it is often used in the sense of singling out a particular method of the constructor from all of the others.

14.1 Outer Constructor Methods

A constructor is just like any other function in Julia in that its overall behavior is defined by the combined behavior of its methods. Accordingly, you can add functionality to a constructor by simply defining new methods. For example, let's say you want to add a constructor method for `Foo` objects that takes only one argument and uses the given value for both the `bar` and `baz` fields. This is simple:

```
julia> Foo(x) = Foo(x,x)
Foo

julia> Foo(1)
Foo(1, 1)
```

You could also add a zero-argument `Foo` constructor method that supplies default values for both of the `bar` and `baz` fields:

```
julia> Foo() = Foo(0)
Foo

julia> Foo()
Foo(0, 0)
```

Here the zero-argument constructor method calls the single-argument constructor method, which in turn calls the automatically provided two-argument constructor method. For reasons that will become clear very shortly, additional constructor methods declared as normal methods like this are called *outer* constructor methods. Outer constructor methods can only ever create a new instance by calling another constructor method, such as the automatically provided default ones.

14.2 Inner Constructor Methods

While outer constructor methods succeed in addressing the problem of providing additional convenience methods for constructing objects, they fail to address the other two use cases mentioned in the introduction of this chapter: enforcing invariants, and allowing construction of self-referential objects. For these problems, one needs *inner* constructor methods. An inner constructor method is like an outer constructor method, except for two differences:

1. It is declared inside the block of a type declaration, rather than outside of it like normal methods.
2. It has access to a special locally existent function called `new` that creates objects of the block's type.

For example, suppose one wants to declare a type that holds a pair of real numbers, subject to the constraint that the first number is not greater than the second one. One could declare it like this:

```
julia> struct OrderedPair
           x::Real
           y::Real
           OrderedPair(x,y) = x > y ? error("out of order") : new(x,y)
       end
```

Now `OrderedPair` objects can only be constructed such that `x <= y`:

```
julia> OrderedPair(1, 2)
OrderedPair(1, 2)

julia> OrderedPair(2,1)
ERROR: out of order
Stacktrace:
 [1] error at ./error.jl:33 [inlined]
 [2] OrderedPair(::Int64, ::Int64) at ./none:4
 [3] top-level scope
```

If the type were declared `mutable`, you could reach in and directly change the field values to violate this invariant. Of course, messing around with an object's internals uninvited is bad practice. You (or someone else) can also provide additional outer constructor methods at any later point, but once a type is declared, there is no way to add more inner constructor methods. Since outer constructor methods can only create objects by calling other constructor methods, ultimately, some inner constructor must be called to create an object. This guarantees that all objects of the declared type must come into existence by a call to one of the inner constructor methods provided with the type, thereby giving some degree of enforcement of a type's invariants.

If any inner constructor method is defined, no default constructor method is provided: it is presumed that you have supplied yourself with all the inner constructors you need. The default constructor is equivalent to writing your own inner constructor method that takes all of the object's fields as parameters (constrained to be of the correct type, if the corresponding field has a type), and passes them to `new`, returning the resulting object:

```
julia> struct Foo
           bar
           baz
           Foo(bar,baz) = new(bar,baz)
       end
```

This declaration has the same effect as the earlier definition of the `Foo` type without an explicit inner constructor method. The following two types are equivalent – one with a default constructor, the other with an explicit constructor:

```
julia> struct T1
           x::Int64
       end

julia> struct T2
           x::Int64
           T2(x) = new(x)
       end

julia> T1(1)
T1(1)
```

```
julia> T2(1)
T2(1)

julia> T1(1.0)
T1(1)

julia> T2(1.0)
T2(1)
```

It is good practice to provide as few inner constructor methods as possible: only those taking all arguments explicitly and enforcing essential error checking and transformation. Additional convenience constructor methods, supplying default values or auxiliary transformations, should be provided as outer constructors that call the inner constructors to do the heavy lifting. This separation is typically quite natural.

14.3 Incomplete Initialization

The final problem which has still not been addressed is construction of self-referential objects, or more generally, recursive data structures. Since the fundamental difficulty may not be immediately obvious, let us briefly explain it. Consider the following recursive type declaration:

```
julia> mutable struct SelfReferential
           obj::SelfReferential
       end
```

This type may appear innocuous enough, until one considers how to construct an instance of it. If a is an instance of `SelfReferential`, then a second instance can be created by the call:

```
julia> b = SelfReferential(a)
```

But how does one construct the first instance when no instance exists to provide as a valid value for its `obj` field? The only solution is to allow creating an incompletely initialized instance of `SelfReferential` with an unassigned `obj` field, and using that incomplete instance as a valid value for the `obj` field of another instance, such as, for example, itself.

To allow for the creation of incompletely initialized objects, Julia allows the `new` function to be called with fewer than the number of fields that the type has, returning an object with the unspecified fields uninitialized. The inner constructor method can then use the incomplete object, finishing its initialization before returning it. Here, for example, is another attempt at defining the `SelfReferential` type, this time using a zero-argument inner constructor returning instances having `obj` fields pointing to themselves:

```
julia> mutable struct SelfReferential
           obj::SelfReferential
           SelfReferential() = (x = new(); x.obj = x)
       end
```

We can verify that this constructor works and constructs objects that are, in fact, self-referential:

```
julia> x = SelfReferential();

julia> x === x
true

julia> x === x.obj
true

julia> x === x.obj.obj
true
```

Although it is generally a good idea to return a fully initialized object from an inner constructor, it is possible to return incompletely initialized objects:

```
julia> mutable struct Incomplete
           data
           Incomplete() = new()
       end

julia> z = Incomplete();
```

While you are allowed to create objects with uninitialized fields, any access to an uninitialized reference is an immediate error:

```
julia> z.data
ERROR: UndefRefError: access to undefined reference
```

This avoids the need to continually check for `null` values. However, not all object fields are references. Julia considers some types to be "plain data", meaning all of their data is self-contained and does not reference other objects. The plain data types consist of primitive types (e.g. `Int`) and immutable structs of other plain data types (see also: `isbits`, `isbitstype`). The initial contents of a plain data type is undefined:

```
julia> struct HasPlain
           n::Int
           HasPlain() = new()
       end

julia> HasPlain()
HasPlain(438103441441)
```

Arrays of plain data types exhibit the same behavior.

You can pass incomplete objects to other functions from inner constructors to delegate their completion:

```
julia> mutable struct Lazy
           data
           Lazy(v) = complete_me(new(), v)
       end
```

As with incomplete objects returned from constructors, if `complete_me` or any of its callees try to access the `data` field of the `Lazy` object before it has been initialized, an error will be thrown immediately.

14.4 Parametric Constructors

Parametric types add a few wrinkles to the constructor story. Recall from Parametric Types that, by default, instances of parametric composite types can be constructed either with explicitly given type parameters or with type parameters implied by the types of the arguments given to the constructor. Here are some examples:

```
julia> struct Point{T<:Real}
           x::T
           y::T
       end

julia> Point(1,2) ## implicit T ##
Point{Int64}(1, 2)

julia> Point(1.0,2.5) ## implicit T ##
Point{Float64}(1.0, 2.5)

julia> Point(1,2.5) ## implicit T ##
ERROR: MethodError: no method matching Point(::Int64, ::Float64)
The type `Point` exists, but no method is defined for this combination of argument types when
↪  trying to construct it.

Closest candidates are:
  Point(::T, ::T) where T<:Real at none:2

julia> Point{Int64}(1, 2) ## explicit T ##
Point{Int64}(1, 2)

julia> Point{Int64}(1.0,2.5) ## explicit T ##
ERROR: InexactError: Int64(2.5)
Stacktrace:
[...]

julia> Point{Float64}(1.0, 2.5) ## explicit T ##
Point{Float64}(1.0, 2.5)

julia> Point{Float64}(1,2) ## explicit T ##
Point{Float64}(1.0, 2.0)
```

As you can see, for constructor calls with explicit type parameters, the arguments are converted to the implied field types: `Point{Int64}(1,2)` works, but `Point{Int64}(1.0,2.5)` raises an `InexactError` when converting `2.5` to `Int64`. When the type is implied by the arguments to the constructor call, as in `Point(1,2)`, then the types of the arguments must agree – otherwise the T cannot be determined – but any pair of real arguments with matching type may be given to the generic `Point` constructor.

What's really going on here is that `Point`, `Point{Float64}` and `Point{Int64}` are all different constructor functions. In fact, `Point{T}` is a distinct constructor function for each type T. Without any explicitly provided inner constructors, the declaration of the composite type `Point{T<:Real}` automatically provides an inner constructor, `Point{T}`, for each possible type `T<:Real`, that behaves just like non-parametric default inner constructors do. It also provides a single general outer `Point` constructor that takes pairs of real arguments, which must be of the same type. This automatic provision of constructors is equivalent to the following explicit declaration:

```
julia> struct Point{T<:Real}
           x::T
           y::T
           Point{T}(x,y) where {T<:Real} = new(x,y)
       end

julia> Point(x::T, y::T) where {T<:Real} = Point{T}(x,y);
```

Notice that each definition looks like the form of constructor call that it handles. The call `Point{Int64}(1,2)` will invoke the definition `Point{T}(x,y)` inside the `struct` block. The outer constructor declaration, on the other hand, defines a method for the general `Point` constructor which only applies to pairs of values of the same real type. This declaration makes constructor calls without explicit type parameters, like `Point(1,2)` and `Point(1.0,2.5)`, work. Since the method declaration restricts the arguments to being of the same type, calls like `Point(1,2.5)`, with arguments of different types, result in "no method" errors.

Suppose we wanted to make the constructor call `Point(1,2.5)` work by "promoting" the integer value `1` to the floating-point value `1.0`. The simplest way to achieve this is to define the following additional outer constructor method:

```
julia> Point(x::Int64, y::Float64) = Point(convert(Float64,x),y);
```

This method uses the `convert` function to explicitly convert `x` to `Float64` and then delegates construction to the general constructor for the case where both arguments are `Float64`. With this method definition what was previously a `MethodError` now successfully creates a point of type `Point{Float64}`:

```
julia> p = Point(1,2.5)
Point{Float64}(1.0, 2.5)

julia> typeof(p)
Point{Float64}
```

However, other similar calls still don't work:

```
julia> Point(1.5,2)
ERROR: MethodError: no method matching Point(::Float64, ::Int64)
The type `Point` exists, but no method is defined for this combination of argument types when
↪  trying to construct it.

Closest candidates are:
  Point(::T, !Matched::T) where T<:Real
   @ Main none:1
  Point(!Matched::Int64, !Matched::Float64)
   @ Main none:1

Stacktrace:
[...]
```

For a more general way to make all such calls work sensibly, see Conversion and Promotion. At the risk of spoiling the suspense, we can reveal here that all it takes is the following outer method definition to make all calls to the general `Point` constructor work as one would expect:

```
julia> Point(x::Real, y::Real) = Point(promote(x,y)...);
```

The `promote` function converts all its arguments to a common type – in this case `Float64`. With this method definition, the `Point` constructor promotes its arguments the same way that numeric operators like `+` do, and works for all kinds of real numbers:

```
julia> Point(1.5,2)
Point{Float64}(1.5, 2.0)

julia> Point(1,1//2)
Point{Rational{Int64}}(1//1, 1//2)

julia> Point(1.0,1//2)
Point{Float64}(1.0, 0.5)
```

Thus, while the implicit type parameter constructors provided by default in Julia are fairly strict, it is possible to make them behave in a more relaxed but sensible manner quite easily. Moreover, since constructors can leverage all of the power of the type system, methods, and multiple dispatch, defining sophisticated behavior is typically quite simple.

14.5 Case Study: Rational

Perhaps the best way to tie all these pieces together is to present a real world example of a parametric composite type and its constructor methods. To that end, we implement our own rational number type `OurRational`, similar to Julia's built-in `Rational` type, defined in `rational.jl`:

```
julia> struct OurRational{T<:Integer} <: Real
           num::T
           den::T
           function OurRational{T}(num::T, den::T) where T<:Integer
               if num == 0 && den == 0
                    error("invalid rational: 0//0")
               end
               num = flipsign(num, den)
               den = flipsign(den, den)
               g = gcd(num, den)
               num = div(num, g)
               den = div(den, g)
               new(num, den)
           end
       end

julia> OurRational(n::T, d::T) where {T<:Integer} = OurRational{T}(n,d)
OurRational

julia> OurRational(n::Integer, d::Integer) = OurRational(promote(n,d)...)
OurRational

julia> OurRational(n::Integer) = OurRational(n,one(n))
OurRational
```

```
julia> ⊘(n::Integer, d::Integer) = OurRational(n,d)
⊘ (generic function with 1 method)

julia> ⊘(x::OurRational, y::Integer) = x.num ⊘ (x.den*y)
⊘ (generic function with 2 methods)

julia> ⊘(x::Integer, y::OurRational) = (x*y.den) ⊘ y.num
⊘ (generic function with 3 methods)

julia> ⊘(x::Complex, y::Real) = complex(real(x) ⊘ y, imag(x) ⊘ y)
⊘ (generic function with 4 methods)

julia> ⊘(x::Real, y::Complex) = (x*y') ⊘ real(y*y')
⊘ (generic function with 5 methods)

julia> function ⊘(x::Complex, y::Complex)
           xy = x*y'
           yy = real(y*y')
           complex(real(xy) ⊘ yy, imag(xy) ⊘ yy)
       end
⊘ (generic function with 6 methods)
```

The first line – `struct OurRational{T<:Integer} <: Real` – declares that `OurRational` takes one type parameter of an integer type, and is itself a real type. The field declarations `num::T` and `den::T` indicate that the data held in a `OurRational{T}` object are a pair of integers of type `T`, one representing the rational value's numerator and the other representing its denominator.

Now things get interesting. `OurRational` has a single inner constructor method which checks that `num` and `den` aren't both zero and ensures that every rational is constructed in "lowest terms" with a non-negative denominator. This is accomplished by first flipping the signs of numerator and denominator if the denominator is negative. Then, both are divided by their greatest common divisor (`gcd` always returns a non-negative number, regardless of the sign of its arguments). Because this is the only inner constructor for `OurRational`, we can be certain that `OurRational` objects are always constructed in this normalized form.

`OurRational` also provides several outer constructor methods for convenience. The first is the "standard" general constructor that infers the type parameter `T` from the type of the numerator and denominator when they have the same type. The second applies when the given numerator and denominator values have different types: it promotes them to a common type and then delegates construction to the outer constructor for arguments of matching type. The third outer constructor turns integer values into rationals by supplying a value of 1 as the denominator.

Following the outer constructor definitions, we defined a number of methods for the `⊘` operator, which provides a syntax for writing rationals (e.g. `1 ⊘ 2`). Julia's `Rational` type uses the `//` operator for this purpose. Before these definitions, `⊘` is a completely undefined operator with only syntax and no meaning. Afterwards, it behaves just as described in Rational Numbers – its entire behavior is defined in these few lines. Note that the infix use of `⊘` works because Julia has a set of symbols that are recognized to be infix operators. The first and most basic definition just makes `a ⊘ b` construct a `OurRational` by applying the `OurRational` constructor to `a` and `b` when they are integers. When one of the operands of `⊘` is already a rational number, we construct a new rational for the resulting ratio slightly differently; this behavior is actually identical to division of a rational with an integer. Finally, applying `⊘` to complex integral values

creates an instance of `Complex{<:OurRational}` – a complex number whose real and imaginary parts are rationals:

```
julia> z = (1 + 2im) ⊘ (1 - 2im);

julia> typeof(z)
Complex{OurRational{Int64}}

julia> typeof(z) <: Complex{<:OurRational}
true
```

Thus, although the `⊘` operator usually returns an instance of `OurRational`, if either of its arguments are complex integers, it will return an instance of `Complex{<:OurRational}` instead. The interested reader should consider perusing the rest of `rational.jl`: it is short, self-contained, and implements an entire basic Julia type.

14.6 Outer-only constructors

As we have seen, a typical parametric type has inner constructors that are called when type parameters are known; e.g. they apply to `Point{Int}` but not to `Point`. Optionally, outer constructors that determine type parameters automatically can be added, for example constructing a `Point{Int}` from the call `Point(1,2)`. Outer constructors call inner constructors to actually make instances. However, in some cases one would rather not provide inner constructors, so that specific type parameters cannot be requested manually.

For example, say we define a type that stores a vector along with an accurate representation of its sum:

```
julia> struct SummedArray{T<:Number,S<:Number}
           data::Vector{T}
           sum::S
       end

julia> SummedArray(Int32[1; 2; 3], Int32(6))
SummedArray{Int32, Int32}(Int32[1, 2, 3], 6)
```

The problem is that we want S to be a larger type than T, so that we can sum many elements with less information loss. For example, when T is `Int32`, we would like S to be `Int64`. Therefore we want to avoid an interface that allows the user to construct instances of the type `SummedArray{Int32,Int32}`. One way to do this is to provide a constructor only for `SummedArray`, but inside the `struct` definition block to suppress generation of default constructors:

```
julia> struct SummedArray{T<:Number,S<:Number}
           data::Vector{T}
           sum::S
           function SummedArray(a::Vector{T}) where T
               S = widen(T)
               new{T,S}(a, sum(S, a))
           end
       end

julia> SummedArray(Int32[1; 2; 3], Int32(6))
```

```
ERROR: MethodError: no method matching SummedArray(::Vector{Int32}, ::Int32)
The type `SummedArray` exists, but no method is defined for this combination of argument types
↪  when trying to construct it.

Closest candidates are:
  SummedArray(::Vector{T}) where T
   @ Main none:4

Stacktrace:
[...]
```

This constructor will be invoked by the syntax `SummedArray(a)`. The syntax `new{T,S}` allows specifying parameters for the type to be constructed, i.e. this call will return a `SummedArray{T,S}`. `new{T,S}` can be used in any constructor definition, but for convenience the parameters to `new{}` are automatically derived from the type being constructed when possible.

14.7 Constructors are just callable objects

An object of any type may be made callable by defining a method. This includes types, i.e., objects of type Type; and constructors may, in fact, be viewed as just callable type objects. For example, there are many methods defined on `Bool` and various supertypes of it:

```
julia> methods(Bool)
# 9 methods for type constructor:
 [1] Bool(x::BigFloat)
     @ mpfr.jl:480
 [2] Bool(x::Float16)
     @ float.jl:360
 [3] Bool(x::Rational)
     @ rational.jl:156
 [4] Bool(x::Real)
     @ bool.jl:191
 [5] (dt::Type{<:Integer})(ip::IPAddr)
     @
     ↪  ~/work/docs.julialang.org/docs.julialang.org/pdf/build/julia-1.13.0-rc4-linux-x86_64/share/julia/stdlib/v1.13/
 [6] (::Type{T})(x::Enum{T2}) where {T<:Integer, T2<:Integer}
     @ Enums.jl:19
 [7] (::Type{T})(z::Complex) where T<:Real
     @ complex.jl:44
 [8] (::Type{T})(x::Base.TwicePrecision) where T<:Number
     @ twiceprecision.jl:265
 [9] (::Type{T})(x::AbstractChar) where T<:Union{AbstractChar, Number}
     @ char.jl:52
```

The usual constructor syntax is exactly equivalent to the function-like object syntax, so trying to define a method with each syntax will cause the first method to be overwritten by the next one:

```
julia> struct S
           f::Int
       end
```

```
julia> S() = S(7)
S

julia> (::Type{S})() = S(8)  # overwrites the previous constructor method

julia> S()
S(8)
```

Chapter 15

Conversion and Promotion

Julia has a system for promoting arguments of mathematical operators to a common type, which has been mentioned in various other sections, including Integers and Floating-Point Numbers, Mathematical Operations and Elementary Functions, Types, and Methods. In this section, we explain how this promotion system works, as well as how to extend it to new types and apply it to functions besides built-in mathematical operators. Traditionally, programming languages fall into two camps with respect to promotion of arithmetic arguments:

- **Automatic promotion for built-in arithmetic types and operators.** In most languages, built-in numeric types, when used as operands to arithmetic operators with infix syntax, such as `+`, `-`, `*`, and `/`, are automatically promoted to a common type to produce the expected results. C, Java, Perl, and Python, to name a few, all correctly compute the sum `1 + 1.5` as the floating-point value `2.5`, even though one of the operands to `+` is an integer. These systems are convenient and designed carefully enough that they are generally all-but-invisible to the programmer: hardly anyone consciously thinks of this promotion taking place when writing such an expression, but compilers and interpreters must perform conversion before addition since integers and floating-point values cannot be added as-is. Complex rules for such automatic conversions are thus inevitably part of specifications and implementations for such languages.
- **No automatic promotion.** This camp includes Ada and ML – very "strict" statically typed languages. In these languages, every conversion must be explicitly specified by the programmer. Thus, the example expression `1 + 1.5` would be a compilation error in both Ada and ML. Instead one must write `real(1) + 1.5`, explicitly converting the integer 1 to a floating-point value before performing addition. Explicit conversion everywhere is so inconvenient, however, that even Ada has some degree of automatic conversion: integer literals are promoted to the expected integer type automatically, and floating-point literals are similarly promoted to appropriate floating-point types.

In a sense, Julia falls into the "no automatic promotion" category: mathematical operators are just functions with special syntax, and the arguments of functions are never automatically converted. However, one may observe that applying mathematical operations to a wide variety of mixed argument types is just an extreme case of polymorphic multiple dispatch – something which Julia's dispatch and type systems are particularly well-suited to handle. "Automatic" promotion of mathematical operands simply emerges as a special application: Julia comes with pre-defined catch-all dispatch rules for mathematical operators, invoked when no specific implementation exists for some combination of operand types. These catch-all rules first promote all operands to a common type using user-definable promotion rules, and then invoke

a specialized implementation of the operator in question for the resulting values, now of the same type. User-defined types can easily participate in this promotion system by defining methods for conversion to and from other types, and providing a handful of promotion rules defining what types they should promote to when mixed with other types.

15.1 Conversion

The standard way to obtain a value of a certain type `T` is to call the type's constructor, `T(x)`. However, there are cases where it's convenient to convert a value from one type to another without the programmer asking for it explicitly. One example is assigning a value into an array: if `A` is a `Vector{Float64}`, the expression `A[1] = 2` should work by automatically converting the 2 from `Int` to `Float64`, and storing the result in the array. This is done via the `convert` function.

The `convert` function generally takes two arguments: the first is a type object and the second is a value to convert to that type. The returned value is the value converted to an instance of given type. The simplest way to understand this function is to see it in action:

```
julia> x = 12
12

julia> typeof(x)
Int64

julia> xu = convert(UInt8, x)
0x0c

julia> typeof(xu)
UInt8

julia> xf = convert(AbstractFloat, x)
12.0

julia> typeof(xf)
Float64

julia> a = Any[1 2 3; 4 5 6]
2×3 Matrix{Any}:
 1  2  3
 4  5  6

julia> convert(Array{Float64}, a)
2×3 Matrix{Float64}:
 1.0  2.0  3.0
 4.0  5.0  6.0
```

Conversion isn't always possible, in which case a `MethodError` is thrown indicating that `convert` doesn't know how to perform the requested conversion:

```
julia> convert(AbstractFloat, "foo")
ERROR: MethodError: Cannot `convert` an object of type String to an object of type AbstractFloat
[...]
```

Some languages consider parsing strings as numbers or formatting numbers as strings to be conversions (many dynamic languages will even perform conversion for you automatically). This is not the case in Julia. Even though some strings can be parsed as numbers, most strings are not valid representations of numbers, and only a very limited subset of them are. Therefore in Julia the dedicated `parse` function must be used to perform this operation, making it more explicit.

When is `convert` called?

The following language constructs call `convert`:

- Assigning to an array converts to the array's element type.
- Assigning to a field of an object converts to the declared type of the field.
- Constructing an object with `new` converts to the object's declared field types.
- Assigning to a variable with a declared type (e.g. `local x::T`) converts to that type.
- A function with a declared return type converts its return value to that type.
- Passing a value to `ccall` converts it to the corresponding argument type.

Conversion vs. Construction

Note that the behavior of `convert(T, x)` appears to be nearly identical to `T(x)`. Indeed, it usually is. However, there is a key semantic difference: since `convert` can be called implicitly, its methods are restricted to cases that are considered "safe" or "unsurprising". `convert` will only convert between types that represent the same basic kind of thing (e.g. different representations of numbers, or different string encodings). It is also usually lossless; converting a value to a different type and back again should result in the exact same value.

There are four general kinds of cases where constructors differ from `convert`:

Constructors for types unrelated to their arguments

Some constructors don't implement the concept of "conversion". For example, `Timer(2)` creates a 2-second timer, which is not really a "conversion" from an integer to a timer.

Mutable collections

`convert(T, x)` is expected to return the original `x` if `x` is already of type `T`. In contrast, if `T` is a mutable collection type then `T(x)` should always make a new collection (copying elements from `x`).

Wrapper types

For some types which "wrap" other values, the constructor may wrap its argument inside a new object even if it is already of the requested type. For example `Some(x)` wraps `x` to indicate that a value is present (in a context where the result might be a `Some` or `nothing`). However, `x` itself might be the object `Some(y)`, in which case the result is `Some(Some(y))`, with two levels of wrapping. `convert(Some, x)`, on the other hand, would just return `x` since it is already a `Some`.

Constructors that don't return instances of their own type

In *very rare* cases it might make sense for the constructor `T(x)` to return an object not of type T. This could happen if a wrapper type is its own inverse (e.g. `Flip(Flip(x)) === x`), or to support an old calling syntax for backwards compatibility when a library is restructured. But `convert(T, x)` should always return a value of type T.

Defining New Conversions

When defining a new type, initially all ways of creating it should be defined as constructors. If it becomes clear that implicit conversion would be useful, and that some constructors meet the above "safety" criteria, then `convert` methods can be added. These methods are typically quite simple, as they only need to call the appropriate constructor. Such a definition might look like this:

```
import Base: convert
convert(::Type{MyType}, x) = MyType(x)
```

The type of the first argument of this method is `Type{MyType}`, the only instance of which is `MyType`. Thus, this method is only invoked when the first argument is the type value `MyType`. Notice the syntax used for the first argument: the argument name is omitted prior to the `::` symbol, and only the type is given. This is the syntax in Julia for a function argument whose type is specified but whose value does not need to be referenced by name.

All instances of some abstract types are by default considered "sufficiently similar" that a universal `convert` definition is provided in Julia Base. For example, this definition states that it's valid to `convert` any `Number` type to any other by calling a 1-argument constructor:

```
convert(::Type{T}, x::Number) where {T<:Number} = T(x)::T
```

This means that new `Number` types only need to define constructors, since this definition will handle `convert` for them. An identity conversion is also provided to handle the case where the argument is already of the requested type:

```
convert(::Type{T}, x::T) where {T<:Number} = x
```

Similar definitions exist for `AbstractString`, `AbstractArray`, and `AbstractDict`.

15.2 Promotion

Promotion refers to converting values of mixed types to a single common type. Although it is not strictly necessary, it is generally implied that the common type to which the values are converted can faithfully represent all of the original values. In this sense, the term "promotion" is appropriate since the values are converted to a "greater" type – i.e. one which can represent all of the input values in a single common type. It is important, however, not to confuse this with object-oriented (structural) super-typing, or Julia's notion of abstract super-types: promotion has nothing to do with the type hierarchy, and everything to do with converting between alternate representations. For instance, although every `Int32` value can also be represented as a `Float64` value, `Int32` is not a subtype of `Float64`.

Promotion to a common "greater" type is performed in Julia by the `promote` function, which takes any number of arguments, and returns a tuple of the same number of values, converted to a common type, or throws an exception if promotion is not possible. The most common use case for promotion is to convert numeric arguments to a common type:

```
julia> promote(1, 2.5)
(1.0, 2.5)

julia> promote(1, 2.5, 3)
(1.0, 2.5, 3.0)

julia> promote(2, 3//4)
(2//1, 3//4)

julia> promote(1, 2.5, 3, 3//4)
(1.0, 2.5, 3.0, 0.75)

julia> promote(1.5, im)
(1.5 + 0.0im, 0.0 + 1.0im)

julia> promote(1 + 2im, 3//4)
(1//1 + 2//1*im, 3//4 + 0//1*im)
```

Floating-point values are promoted to the largest of the floating-point argument types. Integer values are promoted to the largest of the integer argument types. If the types are the same size but differ in signedness, the unsigned type is chosen. Mixtures of integers and floating-point values are promoted to a floating-point type big enough to hold all the values. Integers mixed with rationals are promoted to rationals. Rationals mixed with floats are promoted to floats. Complex values mixed with real values are promoted to the appropriate kind of complex value.

That is really all there is to using promotions. The rest is just a matter of clever application, the most typical "clever" application being the definition of catch-all methods for numeric operations like the arithmetic operators +, -, * and /. Here are some of the catch-all method definitions given in `promotion.jl`:

```
+(x::Number, y::Number) = +(promote(x,y)...)
-(x::Number, y::Number) = -(promote(x,y)...)
*(x::Number, y::Number) = *(promote(x,y)...)
/(x::Number, y::Number) = /(promote(x,y)...)
```

These method definitions say that in the absence of more specific rules for adding, subtracting, multiplying and dividing pairs of numeric values, promote the values to a common type and then try again. That's all there is to it: nowhere else does one ever need to worry about promotion to a common numeric type for arithmetic operations – it just happens automatically. There are definitions of catch-all promotion methods for a number of other arithmetic and mathematical functions in `promotion.jl`, but beyond that, there are hardly any calls to `promote` required in Julia Base. The most common usages of `promote` occur in outer constructors methods, provided for convenience, to allow constructor calls with mixed types to delegate to an inner type with fields promoted to an appropriate common type. For example, recall that `rational.jl` provides the following outer constructor method:

```
Rational(n::Integer, d::Integer) = Rational(promote(n,d)...)
```

This allows calls like the following to work:

```
julia> x = Rational(Int8(15),Int32(-5))
-3//1

julia> typeof(x)
Rational{Int32}
```

For most user-defined types, it is better practice to require programmers to supply the expected types to constructor functions explicitly, but sometimes, especially for numeric problems, it can be convenient to do promotion automatically.

Defining Promotion Rules

Although one could, in principle, define methods for the `promote` function directly, this would require many redundant definitions for all possible permutations of argument types. Instead, the behavior of `promote` is defined in terms of an auxiliary function called `promote_rule`, which one can provide methods for. The `promote_rule` function takes a pair of type objects and returns another type object, such that instances of the argument types will be promoted to the returned type. Thus, by defining the rule:

```
import Base: promote_rule
promote_rule(::Type{Float64}, ::Type{Float32}) = Float64
```

one declares that when 64-bit and 32-bit floating-point values are promoted together, they should be promoted to 64-bit floating-point. The promotion type does not need to be one of the argument types. For example, the following promotion rules both occur in Julia Base:

```
promote_rule(::Type{BigInt}, ::Type{Float64}) = BigFloat
promote_rule(::Type{BigInt}, ::Type{Int8}) = BigInt
```

In the latter case, the result type is `BigInt` since `BigInt` is the only type large enough to hold integers for arbitrary-precision integer arithmetic. Also note that one does not need to define both `promote_rule(::Type{A}, ::Type{B})` and `promote_rule(::Type{B}, ::Type{A})` – the symmetry is implied by the way `promote_rule` is used in the promotion process.

The `promote_rule` function is used as a building block to define a second function called `promote_type`, which, given any number of type objects, returns the common type to which those values, as arguments to `promote` should be promoted. Thus, if one wants to know, in absence of actual values, what type a collection of values of certain types would promote to, one can use `promote_type`:

```
julia> promote_type(Int8, Int64)
Int64
```

Note that we do **not** overload `promote_type` directly: we overload `promote_rule` instead. `promote_type` uses `promote_rule`, and adds the symmetry. Overloading it directly can cause ambiguity errors. We overload `promote_rule` to define how things should be promoted, and we use `promote_type` to query that.

Internally, `promote_type` is used inside of `promote` to determine what type argument values should be converted to for promotion. The curious reader can read the code in `promotion.jl`, which defines the complete promotion mechanism in about 35 lines.

Case Study: Rational Promotions

Finally, we finish off our ongoing case study of Julia's rational number type, which makes relatively sophisticated use of the promotion mechanism with the following promotion rules:

```
import Base: promote_rule
promote_rule(::Type{Rational{T}}, ::Type{S}) where {T<:Integer,S<:Integer} =
↪  Rational{promote_type(T,S)}
promote_rule(::Type{Rational{T}}, ::Type{Rational{S}}) where {T<:Integer,S<:Integer} =
↪  Rational{promote_type(T,S)}
promote_rule(::Type{Rational{T}}, ::Type{S}) where {T<:Integer,S<:AbstractFloat} =
↪  promote_type(T,S)
```

The first rule says that promoting a rational number with any other integer type promotes to a rational type whose numerator/denominator type is the result of promotion of its numerator/denominator type with the other integer type. The second rule applies the same logic to two different types of rational numbers, resulting in a rational of the promotion of their respective numerator/denominator types. The third and final rule dictates that promoting a rational with a float results in the same type as promoting the numerator/denominator type with the float.

This small handful of promotion rules, together with the type's constructors and the default `convert` method for numbers, are sufficient to make rational numbers interoperate completely naturally with all of Julia's other numeric types – integers, floating-point numbers, and complex numbers. By providing appropriate conversion methods and promotion rules in the same manner, any user-defined numeric type can interoperate just as naturally with Julia's predefined numerics.

Chapter 16

Interfaces

A lot of the power and extensibility in Julia comes from a collection of informal interfaces. By extending a few specific methods to work for a custom type, objects of that type not only receive those functionalities, but they are also able to be used in other methods that are written to generically build upon those behaviors.

16.1 Iteration

There are two methods that are always required:

Required method	Brief description
`iterate(iter)`	Returns either a tuple of the first item and initial state or `nothing` if empty
`iterate(iter, state)`	Returns either a tuple of the next item and next state or `nothing` if no items remain

There are several more methods that should be defined in some circumstances. Please note that you should always define at least one of `Base.IteratorSize(IterType)` and `length(iter)` because the default definition of `Base.IteratorSize(IterType)` is `Base.HasLength()`.

Sequential iteration is implemented by the `iterate` function. Instead of mutating objects as they are iterated over, Julia iterators may keep track of the iteration state externally from the object. The return value from iterate is always either a tuple of a value and a state, or `nothing` if no elements remain. The state object will be passed back to the iterate function on the next iteration and is generally considered an implementation detail private to the iterable object.

Any object that defines this function is iterable and can be used in the many functions that rely upon iteration. It can also be used directly in a `for` loop since the syntax:

```
for item in iter   # or  "for item = iter"
    # body
end
```

is translated into:

```
next = iterate(iter)
while next !== nothing
```

Method	When should this method be defined?	Default definition	Brief description
`Base.IteratorSize(IterType)`	If default is not appropriate	`Base.HasLength()`	One of `Base.HasLength()`, `Base.HasShape{N}()`, `Base.IsInfinite()`, or `Base.SizeUnknown()` as appropriate
`length(iter)`	If `Base.IteratorSize()` returns `Base.HasLength()` or `Base.HasShape{N}()`	(*undefined*)	The number of items, if known
`size(iter, [dim])`	If `Base.IteratorSize()` returns `Base.HasShape{N}()`	(*undefined*)	The number of items in each dimension, if known
`Base.IteratorEltype(IterType)`	If default is not appropriate	`Base.HasEltype()`	Either `Base.EltypeUnknown()` or `Base.HasEltype()` as appropriate
`eltype(IterType)`	If default is not appropriate	`Any`	The type of the first entry of the tuple returned by `iterate()`
`Base.isdone(iter, [state])`	**Must** be defined if iterator is stateful	`missing`	Fast-path hint for iterator completion. If not defined for a stateful iterator then functions that check for done-ness, like `isempty()` and `zip()`, may mutate the iterator and cause buggy behaviour!

```
    (item, state) = next
    # body
    next = iterate(iter, state)
end
```

A simple example is an iterable sequence of square numbers with a defined length:

```
julia> struct Squares
           count::Int
       end

julia> Base.iterate(S::Squares, state=1) = state > S.count ? nothing : (state*state, state+1)
```

With only `iterate` definition, the `Squares` type is already pretty powerful. We can iterate over all the elements:

```
julia> for item in Squares(7)
           println(item)
       end
1
4
9
16
```

```
25
36
49
```

We can use many of the builtin methods that work with iterables, like `in` or `sum`:

```
julia> 25 in Squares(10)
true

julia> sum(Squares(100))
338350
```

There are a few more methods we can extend to give Julia more information about this iterable collection. We know that the elements in a `Squares` sequence will always be `Int`. By extending the `eltype` method, we can give that information to Julia and help it make more specialized code in the more complicated methods. We also know the number of elements in our sequence, so we can extend `length`, too:

```
julia> Base.eltype(::Type{Squares}) = Int # Note that this is defined for the type

julia> Base.length(S::Squares) = S.count
```

Now, when we ask Julia to `collect` all the elements into an array it can preallocate a `Vector{Int}` of the right size instead of naively `push!`ing each element into a `Vector{Any}`:

```
julia> collect(Squares(4))
4-element Vector{Int64}:
  1
  4
  9
 16
```

While we can rely upon generic implementations, we can also extend specific methods where we know there is a simpler algorithm. For example, there's a formula to compute the sum of squares, so we can override the generic iterative version with a more performant solution:

```
julia> Base.sum(S::Squares) = (n = S.count; return n*(n+1)*(2n+1)÷6)

julia> sum(Squares(1803))
1955361914
```

This is a very common pattern throughout Julia Base: a small set of required methods define an informal interface that enable many fancier behaviors. In some cases, types will want to additionally specialize those extra behaviors when they know a more efficient algorithm can be used in their specific case.

It is also often useful to allow iteration over a collection in *reverse order* by iterating over `Iterators.reverse(iterator)`. To actually support reverse-order iteration, however, an iterator type T needs to implement `iterate` for `Iterators.Reverse{T}`. (Given `r::Iterators.Reverse{T}`, the underling iterator of type T is `r.itr`.) In our `Squares` example, we would implement `Iterators.Reverse{Squares}` methods:

```
julia> Base.iterate(rS::Iterators.Reverse{Squares}, state=rS.itr.count) = state < 1 ? nothing :
↪  (state*state, state-1)

julia> collect(Iterators.reverse(Squares(4)))
4-element Vector{Int64}:
 16
  9
  4
  1
```

16.2 Indexing

Methods to implement	Brief description
`getindex(X, i)`	`X[i]`, indexed access, non-scalar `i` should allocate a copy
`setindex!(X, v, i)`	`X[i] = v`, indexed assignment
`firstindex(X)`	The first index, used in `X[begin]`
`lastindex(X)`	The last index, used in `X[end]`

For the `Squares` iterable above, we can easily compute the `i`th element of the sequence by squaring it. We can expose this as an indexing expression `S[i]`. To opt into this behavior, `Squares` simply needs to define `getindex`:

```
julia> function Base.getindex(S::Squares, i::Int)
           1 <= i <= S.count || throw(BoundsError(S, i))
           return i*i
       end

julia> Squares(100)[23]
529
```

Additionally, to support the syntax `S[begin]` and `S[end]`, we must define `firstindex` and `lastindex` to specify the first and last valid indices, respectively:

```
julia> Base.firstindex(S::Squares) = 1

julia> Base.lastindex(S::Squares) = length(S)

julia> Squares(23)[end]
529
```

For multi-dimensional begin/end indexing as in `a[3, begin, 7]`, for example, you should define `firstindex(a, dim)` and `lastindex(a, dim)` (which default to calling `first` and `last` on `axes(a, dim)`, respectively).

Note, though, that the above *only* defines `getindex` with one integer index. Indexing with anything other than an `Int` will throw a `MethodError` saying that there was no matching method. In order to support indexing with ranges or vectors of `Int`s, separate methods must be written:

```
julia> Base.getindex(S::Squares, i::Number) = S[convert(Int, i)]
```

```
julia> Base.getindex(S::Squares, I) = [S[i] for i in I]

julia> Squares(10)[[3,4.,5]]
3-element Vector{Int64}:
  9
 16
 25
```

While this is starting to support more of the indexing operations supported by some of the builtin types, there's still quite a number of behaviors missing. This `Squares` sequence is starting to look more and more like a vector as we've added behaviors to it. Instead of defining all these behaviors ourselves, we can officially define it as a subtype of an `AbstractArray`.

16.3 Abstract Arrays

If a type is defined as a subtype of `AbstractArray`, it inherits a very large set of rich behaviors including iteration and multidimensional indexing built on top of single-element access. See the arrays manual page and the Julia Base section for more supported methods.

A key part in defining an `AbstractArray` subtype is `IndexStyle`. Since indexing is such an important part of an array and often occurs in hot loops, it's important to make both indexing and indexed assignment as efficient as possible. Array data structures are typically defined in one of two ways: either it most efficiently accesses its elements using just one index (linear indexing) or it intrinsically accesses the elements with indices specified for every dimension. These two modalities are identified by Julia as `IndexLinear()` and `IndexCartesian()`. Converting a linear index to multiple indexing subscripts is typically very expensive, so this provides a traits-based mechanism to enable efficient generic code for all array types.

This distinction determines which scalar indexing methods the type must define. `IndexLinear()` arrays are simple: just define `getindex(A::ArrayType, i::Int)`. When the array is subsequently indexed with a multidimensional set of indices, the fallback `getindex(A::AbstractArray, I...)` efficiently converts the indices into one linear index and then calls the above method. `IndexCartesian()` arrays, on the other hand, require methods to be defined for each supported dimensionality with `ndims(A)` `Int` indices. For example, `SparseMatrixCSC` from the `SparseArrays` standard library module, only supports two dimensions, so it just defines `getindex(A::SparseMatrixCSC, i::Int, j::Int)`. The same holds for `setindex!`.

Returning to the sequence of squares from above, we could instead define it as a subtype of an `AbstractArray{Int, 1}`:

```
julia> struct SquaresVector <: AbstractArray{Int, 1}
           count::Int
       end

julia> Base.size(S::SquaresVector) = (S.count,)

julia> Base.IndexStyle(::Type{<:SquaresVector}) = IndexLinear()

julia> Base.getindex(S::SquaresVector, i::Int) = i*i
```

Note that it's very important to specify the two parameters of the `AbstractArray`; the first defines the `eltype`, and the second defines the `ndims`. That supertype and those three methods are all it takes for `SquaresVector` to be an iterable, indexable, and completely functional array:

```
julia> s = SquaresVector(4)
4-element SquaresVector:
  1
  4
  9
 16

julia> s[s .> 8]
2-element Vector{Int64}:
  9
 16

julia> s + s
4-element Vector{Int64}:
  2
  8
 18
 32

julia> sin.(s)
4-element Vector{Float64}:
  0.8414709848078965
 -0.7568024953079282
  0.4121184852417566
 -0.2879033166650653
```

As a more complicated example, let's define our own toy N-dimensional sparse-like array type built on top of `Dict`:

```
julia> struct SparseArray{T,N} <: AbstractArray{T,N}
           data::Dict{NTuple{N,Int}, T}
           dims::NTuple{N,Int}
       end

julia> SparseArray(::Type{T}, dims::Int...) where {T} = SparseArray(T, dims);

julia> SparseArray(::Type{T}, dims::NTuple{N,Int}) where {T,N} =
↪  SparseArray{T,N}(Dict{NTuple{N,Int}, T}(), dims);

julia> Base.size(A::SparseArray) = A.dims

julia> Base.similar(A::SparseArray, ::Type{T}, dims::Dims) where {T} = SparseArray(T, dims)

julia> Base.getindex(A::SparseArray{T,N}, I::Vararg{Int,N}) where {T,N} = get(A.data, I, zero(T))

julia> Base.setindex!(A::SparseArray{T,N}, v, I::Vararg{Int,N}) where {T,N} = (A.data[I] = v)
```

Notice that this is an `IndexCartesian` array, so we must manually define `getindex` and `setindex!` at the dimensionality of the array. Unlike the `SquaresVector`, we are able to define `setindex!`, and so we can mutate the array:

```
julia> A = SparseArray(Float64, 3, 3)
```

```
3×3 SparseArray{Float64, 2}:
 0.0  0.0  0.0
 0.0  0.0  0.0
 0.0  0.0  0.0

julia> fill!(A, 2)
3×3 SparseArray{Float64, 2}:
 2.0  2.0  2.0
 2.0  2.0  2.0
 2.0  2.0  2.0

julia> A[:] = 1:length(A); A
3×3 SparseArray{Float64, 2}:
 1.0  4.0  7.0
 2.0  5.0  8.0
 3.0  6.0  9.0
```

The result of indexing an `AbstractArray` can itself be an array (for instance when indexing by an `AbstractRange`). The `AbstractArray` fallback methods use `similar` to allocate an `Array` of the appropriate size and element type, which is filled in using the basic indexing method described above. However, when implementing an array wrapper you often want the result to be wrapped as well:

```
julia> A[1:2,:]
2×3 SparseArray{Float64, 2}:
 1.0  4.0  7.0
 2.0  5.0  8.0
```

In this example it is accomplished by defining `Base.similar(A::SparseArray, ::Type{T}, dims::Dims)` where `T` to create the appropriate wrapped array. (Note that while `similar` supports 1- and 2-argument forms, in most case you only need to specialize the 3-argument form.) For this to work it's important that `SparseArray` is mutable (supports `setindex!`). Defining `similar`, `getindex` and `setindex!` for `SparseArray` also makes it possible to `copy` the array:

```
julia> copy(A)
3×3 SparseArray{Float64, 2}:
 1.0  4.0  7.0
 2.0  5.0  8.0
 3.0  6.0  9.0
```

In addition to all the iterable and indexable methods from above, these types can also interact with each other and use most of the methods defined in Julia Base for `AbstractArrays`:

```
julia> A[SquaresVector(3)]
3-element SparseArray{Float64, 1}:
 1.0
 4.0
 9.0

julia> sum(A)
45.0
```

If you are defining an array type that allows non-traditional indexing (indices that start at something other than 1), you should specialize `axes`. You should also specialize `similar` so that the `dims` argument (ordinarily a `Dims` size-tuple) can accept `AbstractUnitRange` objects, perhaps range-types `Ind` of your own design. For more information, see Arrays with custom indices.

16.4 Strided Arrays

A strided array is a subtype of `AbstractArray` whose entries are stored in memory with fixed strides. Provided the element type of the array is compatible with BLAS, a strided array can utilize BLAS and LAPACK routines for more efficient linear algebra routines. A typical example of a user-defined strided array is one that wraps a standard `Array` with additional structure.

Warning: do not implement these methods if the underlying storage is not actually strided, as it may lead to incorrect results or segmentation faults.

The following function demonstrates how an element at indices `I` in a strided array `A` can be accessed. This function assumes the element type `isbitstype` and the indices are inbounds.

```
julia> function unsafe_strided_getindex(A::AbstractArray{T,N}, I::Vararg{Int, N})::T where {T, N}
           A_cconv = Base.cconvert(Ptr{T}, A)
           GC.@preserve A_cconv begin
               A_ptr = Base.unsafe_convert(Ptr{T}, A_cconv)
               for d in 1:N
                   stride_in_bytes = stride(A, d) * Base.elsize(typeof(A))
                   first_idx = first(axes(A, d))
                   A_ptr += (I[d] - first_idx) * stride_in_bytes
               end
               unsafe_load(A_ptr)
           end
       end;

julia> A = [1 5; 2 6; 3 7; 4 8];

julia> unsafe_strided_getindex(A, 3, 2)
7

julia> V = view(A, 1:2:3, 1:2);

julia> unsafe_strided_getindex(V, 2, 2)
7
```

Here are some examples to demonstrate which type of arrays are strided and which are not:

```
1:5   # not strided (there is no storage associated with this array.)
Vector(1:5)  # is strided with strides (1,)
A = [1 5; 2 6; 3 7; 4 8]  # is strided with strides (1,4)
V = view(A, 1:2, :)   # is strided with strides (1,4)
V = view(A, 1:2:3, 1:2)   # is strided with strides (2,4)
V = view(A, [1,2,4], :)   # is not strided, as the spacing between rows is not fixed.
```

16.5 Customizing broadcasting

Broadcasting is triggered by an explicit call to `broadcast` or `broadcast!`, or implicitly by "dot" operations like `A .+ b` or `f.(x, y)`. Any object that has `axes` and supports indexing can participate as an argument in broadcasting, and by default the result is stored in an `Array`. This basic framework is extensible in three major ways:

- Ensuring that all arguments support broadcast
- Selecting an appropriate output array for the given set of arguments
- Selecting an efficient implementation for the given set of arguments

Not all types support axes and indexing, but many are convenient to allow in broadcast. The `Base.broadcastable` function is called on each argument to broadcast, allowing it to return something different that supports axes and indexing. By default, this is the identity function for all `AbstractArrays` and `Numbers` — they already support axes and indexing.

If a type is intended to act like a "0-dimensional scalar" (a single object) rather than as a container for broadcasting, then the following method should be defined:

```
Base.broadcastable(o::MyType) = Ref(o)
```

that returns the argument wrapped in a 0-dimensional `Ref` container. For example, such a wrapper method is defined for types themselves, functions, special singletons like `missing` and `nothing`, and dates.

Custom array-like types can specialize `Base.broadcastable` to define their shape, but they should follow the convention that `collect(Base.broadcastable(x)) == collect(x)`. A notable exception is `AbstractString`; strings are special-cased to behave as scalars for the purposes of broadcast even though they are iterable collections of their characters (see Strings for more).

The next two steps (selecting the output array and implementation) are dependent upon determining a single answer for a given set of arguments. Broadcast must take all the varied types of its arguments and collapse them down to just one output array and one implementation. Broadcast calls this single answer a "style". Every broadcastable object each has its own preferred style, and a promotion-like system is used to combine these styles into a single answer — the "destination style".

Broadcast Styles

`Base.BroadcastStyle` is the abstract type from which all broadcast styles are derived. When used as a function it has two possible forms, unary (single-argument) and binary. The unary variant states that you intend to implement specific broadcasting behavior and/or output type, and do not wish to rely on the default fallback `Broadcast.DefaultArrayStyle`.

To override these defaults, you can define a custom `BroadcastStyle` for your object:

```
struct MyStyle <: Broadcast.BroadcastStyle end
Base.BroadcastStyle(::Type{<:MyType}) = MyStyle()
```

In some cases it might be convenient not to have to define `MyStyle`, in which case you can leverage one of the general broadcast wrappers:

- `Base.BroadcastStyle(::Type{<:MyType}) = Broadcast.Style{MyType}()` can be used for arbitrary types.
- `Base.BroadcastStyle(::Type{<:MyType}) = Broadcast.ArrayStyle{MyType}()` is preferred if `MyType` is an `AbstractArray`.
- For `AbstractArrays` that only support a certain dimensionality, create a subtype of `Broadcast.AbstractArrayStyle{N}` (see below).

When your broadcast operation involves several arguments, individual argument styles get combined to determine a single `DestStyle` that controls the type of the output container. For more details, see below.

Selecting an appropriate output array

The broadcast style is computed for every broadcasting operation to allow for dispatch and specialization. The actual allocation of the result array is handled by `similar`, using the Broadcasted object as its first argument.

```
Base.similar(bc::Broadcasted{DestStyle}, ::Type{ElType})
```

The fallback definition is

```
similar(bc::Broadcasted{DefaultArrayStyle{N}}, ::Type{ElType}) where {N,ElType} =
    similar(Array{ElType}, axes(bc))
```

However, if needed you can specialize on any or all of these arguments. The final argument `bc` is a lazy representation of a (potentially fused) broadcast operation, a `Broadcasted` object. For these purposes, the most important fields of the wrapper are `f` and `args`, describing the function and argument list, respectively. Note that the argument list can — and often does — include other nested `Broadcasted` wrappers.

For a complete example, let's say you have created a type, `ArrayAndChar`, that stores an array and a single character:

```
struct ArrayAndChar{T,N} <: AbstractArray{T,N}
    data::Array{T,N}
    char::Char
end
Base.size(A::ArrayAndChar) = size(A.data)
Base.getindex(A::ArrayAndChar{T,N}, inds::Vararg{Int,N}) where {T,N} = A.data[inds...]
Base.setindex!(A::ArrayAndChar{T,N}, val, inds::Vararg{Int,N}) where {T,N} = A.data[inds...] =
↪  val
Base.showarg(io::IO, A::ArrayAndChar, toplevel) = print(io, typeof(A), " with char '", A.char,
↪  "'")
```

You might want broadcasting to preserve the `char` "metadata". First we define

```
Base.BroadcastStyle(::Type{<:ArrayAndChar}) = Broadcast.ArrayStyle{ArrayAndChar}()
```

This means we must also define a corresponding `similar` method:

```
function Base.similar(bc::Broadcast.Broadcasted{Broadcast.ArrayStyle{ArrayAndChar}},
↪  ::Type{ElType}) where ElType
    # Scan the inputs for the ArrayAndChar:
    A = find_aac(bc)
    # Use the char field of A to create the output
    ArrayAndChar(similar(Array{ElType}, axes(bc)), A.char)
end

"`A = find_aac(As)` returns the first ArrayAndChar among the arguments."
find_aac(bc::Base.Broadcast.Broadcasted) = find_aac(bc.args)
find_aac(args::Tuple) = find_aac(find_aac(args[1]), Base.tail(args))
find_aac(x) = x
find_aac(::Tuple{}) = nothing
find_aac(a::ArrayAndChar, rest) = a
find_aac(::Any, rest) = find_aac(rest)
```

From these definitions, one obtains the following behavior:

```
julia> a = ArrayAndChar([1 2; 3 4], 'x')
2×2 ArrayAndChar{Int64, 2} with char 'x':
 1  2
 3  4

julia> a .+ 1
2×2 ArrayAndChar{Int64, 2} with char 'x':
 2  3
 4  5

julia> a .+ [5,10]
2×2 ArrayAndChar{Int64, 2} with char 'x':
  6   7
 13  14
```

Extending broadcast with custom implementations

In general, a broadcast operation is represented by a lazy `Broadcasted` container that holds onto the function to be applied alongside its arguments. Those arguments may themselves be more nested `Broadcasted` containers, forming a large expression tree to be evaluated. A nested tree of `Broadcasted` containers is directly constructed by the implicit dot syntax; `5 .+ 2.*x` is transiently represented by `Broadcasted(+, 5, Broadcasted(*, 2, x))`, for example. This is invisible to users as it is immediately realized through a call to `copy`, but it is this container that provides the basis for broadcast's extensibility for authors of custom types. The built-in broadcast machinery will then determine the result type and size based upon the arguments, allocate it, and then finally copy the realization of the `Broadcasted` object into it with a default `copyto!(::AbstractArray, ::Broadcasted)` method. The built-in fallback `broadcast` and `broadcast!` methods similarly construct a transient `Broadcasted` representation of the operation so they can follow the same codepath. This allows custom array implementations to provide their own `copyto!` specialization to customize and optimize broadcasting. This is again determined by the computed broadcast style. This is such an important part of the operation that it is stored as the first type parameter of the `Broadcasted` type, allowing for dispatch and specialization.

For some types, the machinery to "fuse" operations across nested levels of broadcasting is not available or could be done more efficiently incrementally. In such cases, you may need or want to evaluate `x .*`

`(x .+ 1)` as if it had been written `broadcast(*, x, broadcast(+, x, 1))`, where the inner operation is evaluated before tackling the outer operation. This sort of eager operation is directly supported by a bit of indirection; instead of directly constructing `Broadcasted` objects, Julia lowers the fused expression `x .* (x .+ 1)` to `Broadcast.broadcasted(*, x, Broadcast.broadcasted(+, x, 1))`. Now, by default, `broadcasted` just calls the `Broadcasted` constructor to create the lazy representation of the fused expression tree, but you can choose to override it for a particular combination of function and arguments.

As an example, the builtin `AbstractRange` objects use this machinery to optimize pieces of broadcasted expressions that can be eagerly evaluated purely in terms of the start, step, and length (or stop) instead of computing every single element. Just like all the other machinery, `broadcasted` also computes and exposes the combined broadcast style of its arguments, so instead of specializing on `broadcasted(f, args...)`, you can specialize on `broadcasted(::DestStyle, f, args...)` for any combination of style, function, and arguments.

For example, the following definition supports the negation of ranges:

```
broadcasted(::DefaultArrayStyle{1}, ::typeof(-), r::OrdinalRange) = range(-first(r),
↪  step=-step(r), length=length(r))
```

Extending in-place broadcasting

In-place broadcasting can be supported by defining the appropriate `copyto!(dest, bc::Broadcasted)` method. Because you might want to specialize either on `dest` or the specific subtype of `bc`, to avoid ambiguities between packages we recommend the following convention.

If you wish to specialize on a particular style `DestStyle`, define a method for

```
copyto!(dest, bc::Broadcasted{DestStyle})
```

Optionally, with this form you can also specialize on the type of `dest`.

If instead you want to specialize on the destination type `DestType` without specializing on `DestStyle`, then you should define a method with the following signature:

```
copyto!(dest::DestType, bc::Broadcasted{Nothing})
```

This leverages a fallback implementation of `copyto!` that converts the wrapper into a `Broadcasted{Nothing}`. Consequently, specializing on `DestType` has lower precedence than methods that specialize on `DestStyle`.

Similarly, you can completely override out-of-place broadcasting with a `copy(::Broadcasted)` method.

Working with `Broadcasted` objects

In order to implement such a `copy` or `copyto!`, method, of course, you must work with the `Broadcasted` wrapper to compute each element. There are two main ways of doing so:

- `Broadcast.flatten` recomputes the potentially nested operation into a single function and flat list of arguments. You are responsible for implementing the broadcasting shape rules yourself, but this may be helpful in limited situations.
- Iterating over the `CartesianIndices` of the `axes(::Broadcasted)` and using indexing with the resulting `CartesianIndex` object to compute the result.

Writing binary broadcasting rules

The precedence rules are defined by binary `BroadcastStyle` calls:

```
Base.BroadcastStyle(::Style1, ::Style2) = Style12()
```

where `Style12` is the `BroadcastStyle` you want to choose for outputs involving arguments of `Style1` and `Style2`. For example,

```
Base.BroadcastStyle(::Broadcast.Style{Tuple}, ::Broadcast.AbstractArrayStyle{0}) =
↪  Broadcast.Style{Tuple}()
```

indicates that `Tuple` "wins" over zero-dimensional arrays (the output container will be a tuple). It is worth noting that you do not need to (and should not) define both argument orders of this call; defining one is sufficient no matter what order the user supplies the arguments in.

For `AbstractArray` types, defining a `BroadcastStyle` supersedes the fallback choice, `Broadcast.DefaultArrayStyle`. `DefaultArrayStyle` and the abstract supertype, `AbstractArrayStyle`, store the dimensionality as a type parameter to support specialized array types that have fixed dimensionality requirements.

`DefaultArrayStyle` "loses" to any other `AbstractArrayStyle` that has been defined because of the following methods:

```
BroadcastStyle(a::AbstractArrayStyle{Any}, ::DefaultArrayStyle) = a
BroadcastStyle(a::AbstractArrayStyle{N}, ::DefaultArrayStyle{N}) where N = a
BroadcastStyle(a::AbstractArrayStyle{M}, ::DefaultArrayStyle{N}) where {M,N} =
    typeof(a)(Val(max(M, N)))
```

You do not need to write binary `BroadcastStyle` rules unless you want to establish precedence for two or more non-`DefaultArrayStyle` types.

If your array type does have fixed dimensionality requirements, then you should subtype `AbstractArrayStyle`. For example, the sparse array code has the following definitions:

```
struct SparseVecStyle <: Broadcast.AbstractArrayStyle{1} end
struct SparseMatStyle <: Broadcast.AbstractArrayStyle{2} end
Base.BroadcastStyle(::Type{<:SparseVector}) = SparseVecStyle()
Base.BroadcastStyle(::Type{<:SparseMatrixCSC}) = SparseMatStyle()
```

Whenever you subtype `AbstractArrayStyle`, you also need to define rules for combining dimensionalities, by creating a constructor for your style that takes a `Val(N)` argument. For example:

```
SparseVecStyle(::Val{0}) = SparseVecStyle()
SparseVecStyle(::Val{1}) = SparseVecStyle()
SparseVecStyle(::Val{2}) = SparseMatStyle()
SparseVecStyle(::Val{N}) where N = Broadcast.DefaultArrayStyle{N}()
```

These rules indicate that the combination of a `SparseVecStyle` with 0- or 1-dimensional arrays yields another `SparseVecStyle`, that its combination with a 2-dimensional array yields a `SparseMatStyle`, and anything of higher dimensionality falls back to the dense arbitrary-dimensional framework. These rules allow broadcasting to keep the sparse representation for operations that result in one or two dimensional outputs, but produce an `Array` for any other dimensionality.

16.6 Instance Properties

Sometimes, it is desirable to change how the end-user interacts with the fields of an object. Instead of granting direct access to type fields, an extra layer of abstraction between the user and the code can be provided by overloading `object.field`. Properties are what the user *sees of* the object, fields what the object *actually is*.

By default, properties and fields are the same. However, this behavior can be changed. For example, take this representation of a point in a plane in polar coordinates:

```
julia> mutable struct Point
           r::Float64
           ϕ::Float64
       end

julia> p = Point(7.0, pi/4)
Point(7.0, 0.7853981633974483)
```

As described in the table above dot access `p.r` is the same as `getproperty(p, :r)` which is by default the same as `getfield(p, :r)`:

```
julia> propertynames(p)
(:r, :ϕ)

julia> getproperty(p, :r), getproperty(p, :ϕ)
(7.0, 0.7853981633974483)

julia> p.r, p.ϕ
(7.0, 0.7853981633974483)

julia> getfield(p, :r), getproperty(p, :ϕ)
(7.0, 0.7853981633974483)
```

However, we may want users to be unaware that `Point` stores the coordinates as `r` and `ϕ` (fields), and instead interact with `x` and `y` (properties). The methods in the first column can be defined to add new functionality:

```
julia> Base.propertynames(::Point, private::Bool=false) = private ? (:x, :y, :r, :ϕ) : (:x, :y)

julia> function Base.getproperty(p::Point, s::Symbol)
           if s === :x
               return getfield(p, :r) * cos(getfield(p, :ϕ))
           elseif s === :y
               return getfield(p, :r) * sin(getfield(p, :ϕ))
           else
               # This allows accessing fields with p.r and p.ϕ
               return getfield(p, s)
           end
       end

julia> function Base.setproperty!(p::Point, s::Symbol, f)
```

```
        if s === :x
            y = p.y
            setfield!(p, :r, sqrt(f^2 + y^2))
            setfield!(p, :ϕ, atan(y, f))
            return f
        elseif s === :y
            x = p.x
            setfield!(p, :r, sqrt(x^2 + f^2))
            setfield!(p, :ϕ, atan(f, x))
            return f
        else
            # This allow modifying fields with p.r and p.ϕ
            return setfield!(p, s, f)
        end
    end
```

It is important that `getfield` and `setfield` are used inside `getproperty` and `setproperty!` instead of the dot syntax, since the dot syntax would make the functions recursive which can lead to type inference issues. We can now try out the new functionality:

```
julia> propertynames(p)
(:x, :y)

julia> p.x
4.949747468305833

julia> p.y = 4.0
4.0

julia> p.r
6.363961030678928
```

Finally, it is worth noting that adding instance properties like this is quite rarely done in Julia and should in general only be done if there is a good reason for doing so.

16.7 Rounding

To support rounding on a new type it is typically sufficient to define the single method `round(x::ObjType, r::RoundingMode)`. The passed rounding mode determines in which direction the value should be rounded. The most commonly used rounding modes are `RoundNearest`, `RoundToZero`, `RoundDown`, and `RoundUp`, as these rounding modes are used in the definitions of the one argument `round`, method, and `trunc`, `floor`, and `ceil`, respectively.

In some cases, it is possible to define a three-argument `round` method that is more accurate or performant than the two-argument method followed by conversion. In this case it is acceptable to define the three argument method in addition to the two argument method. If it is impossible to represent the rounded result as an object of the type `T`, then the three argument method should throw an `InexactError`.

For example, if we have an `Interval` type which represents a range of possible values similar to https://github.com/JuliaPhysics/Measurements.jl, we may define rounding on that type with the following

```
julia> struct Interval{T}
           min::T
           max::T
       end

julia> Base.round(x::Interval, r::RoundingMode) = Interval(round(x.min, r), round(x.max, r))

julia> x = Interval(1.7, 2.2)
Interval{Float64}(1.7, 2.2)

julia> round(x)
Interval{Float64}(2.0, 2.0)

julia> floor(x)
Interval{Float64}(1.0, 2.0)

julia> ceil(x)
Interval{Float64}(2.0, 3.0)

julia> trunc(x)
Interval{Float64}(1.0, 2.0)
```

Methods to implement		Brief description
`size(A)`		Returns a tuple containing the dimensions of `A`
`getindex(A, i::Int)`		(if `IndexLinear`) Linear scalar indexing
`getindex(A, I::Vararg{Int, N})`		(if `IndexCartesian`, where `N = ndims(A)`) N-dimensional scalar indexing
Optional methods	**Default definition**	**Brief description**
`IndexStyle(::Type)`	`IndexCartesian()`	Returns either `IndexLinear()` or `IndexCartesian()`. See the description below.
`setindex!(A, v, i::Int)`		(if `IndexLinear`) Scalar indexed assignment
`setindex!(A, v, I::Vararg{Int, N})`		(if `IndexCartesian`, where `N = ndims(A)`) N-dimensional scalar indexed assignment
`getindex(A, I...)`	defined in terms of scalar `getindex`	Multidimensional and nonscalar indexing
`setindex!(A, X, I...)`	defined in terms of scalar `setindex!`	Multidimensional and nonscalar indexed assignment
`iterate`	defined in terms of scalar `getindex`	Iteration
`length(A)`	`prod(size(A))`	Number of elements
`similar(A)`	`similar(A, eltype(A), size(A))`	Return a mutable array with the same shape and element type
`similar(A, ::Type{S})`	`similar(A, S, size(A))`	Return a mutable array with the same shape and the specified element type
`similar(A, dims::Dims)`	`similar(A, eltype(A), dims)`	Return a mutable array with the same element type and size *dims*
`similar(A, ::Type{S}, dims::Dims)`	`Array{S}(undef, dims)`	Return a mutable array with the specified element type and size
Non-traditional indices	**Default definition**	**Brief description**
`axes(A)`	`map(OneTo, size(A))`	Return a tuple of `AbstractUnitRange{<:Integer}` of valid indices. The axes should be their own axes, that is `axes.(axes(A),1) == axes(A)` should be satisfied.
`similar(A, ::Type{S}, inds)`	`similar(A, S, Base.to_shape(inds))`	Return a mutable array with the specified indices `inds` (see below)
`similar(T::Union{Type,Function}, inds)`	`T(Base.to_shape(inds))`	Return an array similar to `T` with the specified indices `inds` (see below)

Methods to implement	Brief description
`strides(A)`	Return the distance in memory (in number of elements) between adjacent elements in each dimension as a tuple. If `A` is an `AbstractArray{T,0}`, this should return an empty tuple.
`Base.unsafe_convert(::Type{Ptr{T}}, A)`	Return the native address of an array.
`Base.elsize(::Type{<:A})`	Return the stride (in number of bytes) between consecutive elements in the array.
Optional methods	**Default definition**	**Brief description**
`stride(A, i::Int)`	`strides(A)[i]`	Return the distance in memory (in number of elements) between adjacent elements in dimension `i`.

Methods to implement	Brief description
`Base.BroadcastStyle(::Type{SrcType}) = SrcStyle()`	Broadcasting behavior of `SrcType`
`Base.similar(bc::Broadcasted{DestStyle}, ::Type{ElType})`	Allocation of output container
Optional methods	
`Base.BroadcastStyle(::Style1, ::Style2) = Style12()`	Precedence rules for mixing styles
`Base.axes(x)`	Declaration of the indices of `x`, as per `axes(x)`.
`Base.broadcastable(x)`	Convert `x` to an object that has `axes` and supports indexing
Bypassing default machinery	
`Base.copy(bc::Broadcasted{DestStyle})`	Custom implementation of `broadcast`
`Base.copyto!(dest, bc::Broadcasted{DestStyle})`	Custom implementation of `broadcast!`, specializing on `DestStyle`
`Base.copyto!(dest::DestType, bc::Broadcasted{Nothing})`	Custom implementation of `broadcast!`, specializing on `DestType`
`Base.Broadcast.broadcasted(f, args...)`	Override the default lazy behavior within a fused expression
`Base.Broadcast.instantiate(bc::Broadcasted{DestStyle})`	Override the computation of the lazy broadcast's axes

Methods to implement	Default definition	Brief description
`propertynames(x::ObjType, private::Bool=false)`	`fieldnames(typeof(x))`	Return a tuple of the properties (`x.property`) of an object `x`. If `private=true`, also return property names intended to be kept as private
`getproperty(x::ObjType, s::Symbol)`	`getfield(x, s)`	Return property `s` of `x`. `x.s` calls `getproperty(x, :s)`.
`setproperty!(x::ObjType, s::Symbol, v)`	`setfield!(x, s, v)`	Set property `s` of `x` to `v`. `x.s = v` calls `setproperty!(x, :s, v)`. Should return `v`.

Methods to implement	Default definition	Brief description
`round(x::ObjType, r::RoundingMode)`	none	Round x and return the result. If possible, round should return an object of the same type as x
`round(T::Type, x::ObjType, r::RoundingMode)`	`convert(T, round(x, r))`	Round x, returning the result as a T

Chapter 17

Modules

Modules in Julia help organize code into coherent units. They are delimited syntactically inside `module NameOfModule ... end`, and have the following features:

1. Modules are separate namespaces, each introducing a new global scope. This is useful, because it allows the same name to be used for different functions or global variables without conflict, as long as they are in separate modules.
2. Modules have facilities for detailed namespace management: each defines a set of names it `exports` and marks as `public`, and can import names from other modules with `using` and `import` (we explain these below).
3. Modules can be precompiled for faster loading, and may contain code for runtime initialization.

Typically, in larger Julia packages you will see module code organized into files, eg

```
module SomeModule

# export, public, using, import statements are usually here; we discuss these below

include("file1.jl")
include("file2.jl")

end
```

Files and file names are mostly unrelated to modules; modules are associated only with module expressions. One can have multiple files per module, and multiple modules per file. `include` behaves as if the contents of the source file were evaluated in the global scope of the including module. In this chapter, we use short and simplified examples, so we won't use `include`.

The recommended style is not to indent the body of the module, since that would typically lead to whole files being indented. Also, it is common to use `UpperCamelCase` for module names (just like types), and use the plural form if applicable, especially if the module contains a similarly named identifier, to avoid name clashes. For example,

```
module FastThings

struct FastThing
    ...
end

end
```

17.1 Namespace management

Namespace management refers to the facilities the language offers for making names in a module available in other modules. We discuss the related concepts and functionality below in detail.

Qualified names

Names for functions, variables and types in the global scope like `sin`, `ARGS`, and `UnitRange` always belong to a module, called the *parent module*, which can be found interactively with `parentmodule`, for example

```
julia> parentmodule(UnitRange)
Base
```

One can also refer to these names outside their parent module by prefixing them with their module, eg `Base.UnitRange`. This is called a *qualified name*. The parent module may be accessible using a chain of submodules like `Base.Math.sin`, where `Base.Math` is called the *module path*. Due to syntactic ambiguities, qualifying a name that contains only symbols, such as an operator, requires inserting a colon, e.g. `Base.:+`. A small number of operators additionally require parentheses, e.g. `Base.:(==)`.

If a name is qualified, then it is always *accessible*, and in case of a function, it can also have methods added to it by using the qualified name as the function name.

Within a module, a variable name can be "reserved" without assigning to it by declaring it as `global x`. This prevents name conflicts for globals initialized after load time. The syntax `M.x = y` does not work to assign a global in another module; global assignment is always module-local.

Export lists

Names (referring to functions, types, global variables, and constants) can be added to the *export list* of a module with `export`: these are the symbols that are imported when `using` the module. Typically, they are at or near the top of the module definition so that readers of the source code can find them easily, as in

```
julia> module NiceStuff
       export nice, DOG
       struct Dog end      # singleton type, not exported
       const DOG = Dog()   # named instance, exported
       nice(x) = "nice $x" # function, exported
       end;
```

but this is just a style suggestion — a module can have multiple `export` statements in arbitrary locations.

It is common to export names which form part of the API (application programming interface). In the above code, the export list suggests that users should use `nice` and `DOG`. However, since qualified names always make identifiers accessible, this is just an option for organizing APIs: unlike other languages, Julia has no facilities for truly hiding module internals.

Also, some modules don't export names at all. This is usually done if they use common words, such as `derivative`, in their API, which could easily clash with the export lists of other modules. We will see how to manage name clashes below.

To mark a name as public without exporting it into the namespace of folks who call `using NiceStuff`, one can use `public` instead of `export`. This marks the public name(s) as part of the public API, but does not have any namespace implications. The `public` keyword is only available in Julia 1.11 and above. To maintain compatibility with Julia 1.10 and below, use the `@compat` macro from the Compat package, or the version-aware construct

```
VERSION >= v"1.11.0-DEV.469" && eval(Meta.parse("public a, b, c"))
```

`export` is a keyword wherever it occurs whereas the `public` keyword is currently limited to the syntactic top level within a file or module. This limitation exists for compatibility reasons, as `public` was introduced as a new keyword in Julia 1.11 while `export` has existed since Julia 1.0. However, this restriction on `public` may be lifted in future releases, so do not use `public` as an identifier.

Standalone `using` and `import`

For interactive use, the most common way of loading a module is `using ModuleName`. This loads the code associated with `ModuleName`, and brings

1. the module name
2. and the elements of the export list into the surrounding global namespace.

Technically, the statement `using ModuleName` means that a module called `ModuleName` will be available for resolving names as needed. When a global variable is encountered that has no definition in the current module, the system will search for it among variables exported by `ModuleName` and use it if it is found there. This means that all uses of that global within the current module will resolve to the definition of that variable in `ModuleName`.

To load a module from a package, the statement `using ModuleName` can be used. To load a module from a locally defined module, a dot needs to be added before the module name like `using .ModuleName`.

To continue with our example,

```
julia> using .NiceStuff
```

would load the above code, making `NiceStuff` (the module name), `DOG` and `nice` available. `Dog` is not on the export list, but it can be accessed if the name is qualified with the module path (which here is just the module name) as `NiceStuff.Dog`.

Importantly, **`using ModuleName` is the only form for which export lists matter at all**.

In contrast,

```
julia> import .NiceStuff
```

brings *only* the module name into scope. Users would need to use `NiceStuff.DOG`, `NiceStuff.Dog`, and `NiceStuff.nice` to access its contents. As we will see in the next section `import .NiceStuff` is equivalent to `using .NiceStuff: NiceStuff`. Usually, `import ModuleName` or `using ModuleName: ModuleName` is used in contexts when the user wants to keep the namespace clean.

You can combine multiple `using` and `import` statements of the same kind in a comma-separated expression, e.g.

```
julia> using LinearAlgebra, Random
```

`using` and `import` with specific identifiers, and adding methods

When `using ModuleName:` or `import ModuleName:` is followed by a comma-separated list of names, the module is loaded, but *only those specific names are brought into the namespace* by the statement. For example,

```
julia> using .NiceStuff: nice, DOG
```

will import the names `nice` and `DOG`.

Importantly, the module name `NiceStuff` will *not* be in the namespace. If you want to make it accessible, you have to list it explicitly, as

```
julia> using .NiceStuff: nice, DOG, NiceStuff
```

When two or more packages/modules export a name and that name does not refer to the same thing in each of the packages, and the packages are loaded via `using` without an explicit list of names, it is an error to reference that name without qualification. It is thus recommended that code intended to be forward-compatible with future versions of its dependencies and of Julia, e.g., code in released packages, list the names it uses from each loaded package, e.g., `using Foo: Foo, f` rather than `using Foo`.

Julia has two forms for seemingly the same thing because only `import ModuleName: f` allows adding methods to `f` *without a module path*. That is to say, the following example will give an error:

```
julia> using .NiceStuff: nice

julia> struct Cat end

julia> nice(::Cat) = "nice "
ERROR: invalid method definition in Main: function NiceStuff.nice must be explicitly imported to
↪  be extended
Stacktrace:
 [1] top-level scope
   @ none:1
```

This error prevents accidentally adding methods to functions in other modules that you only intended to use.

There are two ways to deal with this. You can always qualify function names with a module path:

```
julia> using .NiceStuff

julia> struct Cat end

julia> NiceStuff.nice(::Cat) = "nice 😺"
```

Alternatively, you can `import` the specific function name:

```
julia> import .NiceStuff: nice

julia> struct Mouse end

julia> nice(::Mouse) = "nice 🐭"
nice (generic function with 3 methods)
```

Which one you choose is a matter of style. The first form makes it clear that you are adding a method to a function in another module (remember, that the imports and the method definition may be in separate files), while the second one is shorter, which is especially convenient if you are defining multiple methods.

Once a variable is made visible via `using` or `import`, a module may not create its own variable with the same name. Imported variables are read-only; assigning to a global variable always affects a variable owned by the current module, or else raises an error.

Renaming with as

An identifier brought into scope by `import` or `using` can be renamed with the keyword `as`. This is useful for working around name conflicts as well as for shortening names. For example, `Base` exports the function name `read`, but the CSV.jl package also provides `CSV.read`. If we are going to invoke CSV reading many times, it would be convenient to drop the `CSV.` qualifier. But then it is ambiguous whether we are referring to `Base.read` or `CSV.read`:

```
julia> read;

julia> import CSV: read
WARNING: ignoring conflicting import of CSV.read into Main
```

Renaming provides a solution:

```
julia> import CSV: read as rd
```

Imported packages themselves can also be renamed:

```
import BenchmarkTools as BT
```

`as` works with `using` only when a single identifier is brought into scope. For example `using CSV: read as rd` works, but `using CSV as C` does not, since it operates on all of the exported names in `CSV`.

Mixing multiple `using` and `import` statements

When multiple `using` or `import` statements of any of the forms above are used, their effect is combined in the order they appear. For example,

```
julia> using .NiceStuff         # exported names and the module name

julia> import .NiceStuff: nice  # allows adding methods to unqualified functions
```

would bring all the exported names of `NiceStuff` and the module name itself into scope, and also allow adding methods to `nice` without prefixing it with a module name.

Handling name conflicts

Consider the situation where two (or more) packages export the same name, as in

```
julia> module A
       export f
       f() = 1
       end
A
julia> module B
       export f
       f() = 2
       end
B
```

The statement `using .A, .B` works, but when you try to call `f`, you get an error with a hint

```
julia> using .A, .B

julia> f
ERROR: UndefVarError: `f` not defined in `Main`
Hint: It looks like two or more modules export different bindings with this name, resulting in
↪  ambiguity. Try explicitly importing it from a particular module, or qualifying the name with
↪  the module it should come from.
```

Here, Julia cannot decide which `f` you are referring to, so you have to make a choice. The following solutions are commonly used:

1. Simply proceed with qualified names like `A.f` and `B.f`. This makes the context clear to the reader of your code, especially if `f` just happens to coincide but has different meaning in various packages. For example, degree has various uses in mathematics, the natural sciences, and in everyday life, and these meanings should be kept separate.
2. Use the `as` keyword above to rename one or both identifiers, eg

   ```
   julia> using .A: f as f

   julia> using .B: f as g
   ```

would make `B.f` available as `g`. Here, we are assuming that you did not use `using A` before, which would have brought `f` into the namespace.

3. When the names in question *do* share a meaning, it is common for one module to import it from another, or have a lightweight "base" package with the sole function of defining an interface like this, which can be used by other packages. It is conventional to have such package names end in `...Base` (which has nothing to do with Julia's `Base` module).

Precedence order of definitions

There are in general four kinds of binding definitions:

1. Those provided via implicit import through `using M`
2. Those provided via explicit import (e.g. `using M: x`, `import M: x`)
3. Those declared implicitly as global (via `global x` without type specification)
4. Those declared explicitly using definition syntax (`const`, `global x::T`, `struct`, etc.)

Syntactically, we divide these into three precedence levels (from weakest to strongest)

1. Implicit imports
2. Implicit declarations
3. Explicit declarations and imports

In general, we permit replacement of weaker bindings by stronger ones:

```
julia> module M1; const x = 1; export x; end
Main.M1

julia> using .M1

julia> x # Implicit import from M1
1

julia> begin; f() = (global x; x = 1) end

julia> x # Implicit declaration
ERROR: UndefVarError: `x` not defined in `Main`
Suggestion: add an appropriate import or assignment. This global was declared but not assigned.

julia> const x = 2 # Explicit declaration
2
```

However, within the explicit precedence level, replacement is syntactically disallowed:

```
julia> module M1; const x = 1; export x; end
Main.M1

julia> import .M1: x

julia> const x = 2
ERROR: cannot declare Main.x constant; it was already declared as an import
Stacktrace:
 [1] top-level scope
   @ REPL[3]:1
```

or ignored:

```
julia> const y = 2
2

julia> import .M1: x as y
WARNING: import of M1.x into Main conflicts with an existing identifier; ignored.
```

The resolution of an implicit binding depends on the set of all `using`'d modules visible in the current world age. See the manual chapter on world age for more details.

Default top-level definitions and bare modules

Modules automatically contain `using Core`, `using Base`, and definitions of the `eval` and `include` functions, which evaluate expressions/files within the global scope of that module.

If these default definitions are not wanted, modules can be defined using the keyword `baremodule` instead (note: `Core` is still imported). In terms of `baremodule`, a standard `module` looks like this:

```
baremodule Mod

using Base

eval(x) = Core.eval(Mod, x)
include(p) = Base.include(Mod, p)

...

end
```

If even `Core` is not wanted, a module that imports nothing and defines no names at all can be defined with `Module(:YourNameHere, false, false)` and code can be evaluated into it with `@eval` or `Core.eval`:

```
julia> arithmetic = Module(:arithmetic, false, false)
Main.arithmetic

julia> @eval arithmetic add(x, y) = $(+)(x, y)
add (generic function with 1 method)

julia> arithmetic.add(12, 13)
25
```

Standard modules

There are three important standard modules:

- `Core` contains all functionality "built into" the language.
- `Base` contains basic functionality that is useful in almost all cases.
- `Main` is the top-level module and the current module, when Julia is started.

Standard library modules

By default Julia ships with some standard library modules. These behave like regular Julia packages except that you don't need to install them explicitly. For example, if you wanted to perform some unit testing, you could load the `Test` standard library as follows:

```
using Test
```

17.2 Submodules and relative paths

Modules can contain *submodules*, nesting the same syntax `module ... end`. They can be used to introduce separate namespaces, which can be helpful for organizing complex codebases. Note that each `module` introduces its own scope, so submodules do not automatically "inherit" names from their parent.

It is recommended that submodules refer to other modules within the enclosing parent module (including the latter) using *relative module qualifiers* in `using` and `import` statements. A relative module qualifier starts with a period (`.`), which corresponds to the current module, and each successive `.` leads to the parent of the current module. This should be followed by modules if necessary, and eventually the actual name to access, all separated by `.`s. As a special case, however, referring to the module root can be written without `.`, avoiding the need to count the depth to reach that module.

Consider the following example, where the submodule SubA defines a function, which is then extended in its "sibling" module:

```
julia> module ParentModule
       module SubA
       export add_D  # exported interface
       const D = 3
       add_D(x) = x + D
       end
       using .SubA  # brings `add_D` into the namespace
       export add_D # export it from ParentModule too
       module SubB
       import ..SubA: add_D # relative path for a "sibling" module
       # import ParentModule.SubA: add_D # when in a package, such as when this is loaded by
       ↪  using or import, this would be equivalent to the previous import, but not at the REPL
       struct Infinity end
       add_D(x::Infinity) = x
       end
       end;
```

You may see code in packages, which, in a similar situation, uses import without the `.`:

```
julia> import ParentModule.SubA: add_D
ERROR: ArgumentError: Package ParentModule not found in current path.
```

However, since this operates through code loading, it only works if `ParentModule` is in a package in a file. If `ParentModule` was defined at the REPL, it is necessary to use use relative paths:

```
julia> import .ParentModule.SubA: add_D
```

Note that the order of definitions also matters if you are evaluating values. Consider

```
module TestPackage

export x, y

x = 0

module Sub
using ..TestPackage
z = y # ERROR: UndefVarError: `y` not defined in `Main`
end

y = 1

end
```

where Sub is trying to use `TestPackage.y` before it was defined, so it does not have a value.

For similar reasons, you cannot use a cyclic ordering:

```
module A

module B
using ..C # ERROR: UndefVarError: `C` not defined in `Main.A`
end

module C
using ..B
end

end
```

17.3 Module initialization and precompilation

Large modules can take several seconds to load because executing all of the statements in a module often involves compiling a large amount of code. Julia creates precompiled caches of the module to reduce this time.

Precompiled module files (sometimes called "cache files") are created and used automatically when `import` or `using` loads a module. If the cache file(s) do not yet exist, the module will be compiled and saved for future reuse. You can also manually call `Base.compilecache(Base.identify_package("modulename"))` to create these files without loading the module. The resulting cache files will be stored in the `compiled` subfolder of `DEPOT_PATH[1]`. If nothing about your system changes, such cache files will be used when you load the module with `import` or `using`.

Precompilation cache files store definitions of modules, types, methods, and constants. They may also store method specializations and the code generated for them, but this typically requires that the developer add explicit `precompile` directives or execute workloads that force compilation during the package build.

However, if you update the module's dependencies or change its source code, the module is automatically recompiled upon `using` or `import`. Dependencies are modules it imports, the Julia build, files it includes, or explicit dependencies declared by `include_dependency(path)` in the module file(s).

For file dependencies loaded by `include`, a change is determined by examining whether the file size (`fsize`) or content (condensed into a hash) is unchanged. For file dependencies loaded by `include_dependency` a change is determined by examining whether the modification time (`mtime`) is unchanged, or equal to the modification time truncated to the nearest second (to accommodate systems that can't copy mtime with sub-second accuracy). It also takes into account whether the path to the file chosen by the search logic in `require` matches the path that had created the precompile file. It also takes into account the set of dependencies already loaded into the current process and won't recompile those modules, even if their files change or disappear, in order to avoid creating incompatibilities between the running system and the precompile cache. Finally, it takes account of changes in any compile-time preferences.

If you know that a module is *not* safe to precompile (for example, for one of the reasons described below), you should put `__precompile__(false)` in the module file (typically placed at the top). This will cause `Base.compilecache` to throw an error, and will cause `using` / `import` to load it directly into the current process and skip the precompile and caching. This also thereby prevents the module from being imported by any other precompiled module.

You may need to be aware of certain behaviors inherent in the creation of incremental shared libraries which may require care when writing your module. For example, external state is not preserved. To accommodate this, explicitly separate any initialization steps that must occur at *runtime* from steps that can occur at *compile time*. For this purpose, Julia allows you to define an `__init__()` function in your module that executes any initialization steps that must occur at runtime. This function will not be called during compilation (`--output-*`). Effectively, you can assume it will be run exactly once in the lifetime of the code. You may, of course, call it manually if necessary, but the default is to assume this function deals with computing state for the local machine, which does not need to be – or even should not be – captured in the compiled image. It will be called after the module is loaded into a process, including if it is being loaded into an incremental compile (`--output-incremental=yes`), but not if it is being loaded into a full-compilation process.

In particular, if you define a `function __init__()` in a module, then Julia will call `__init__()` immediately *after* the module is loaded (e.g., by `import`, `using`, or `require`) at runtime for the *first* time (i.e., `__init__` is only called once, and only after all statements in the module have been executed). Because it is called after the module is fully imported, any submodules or other imported modules have their `__init__` functions called *before* the `__init__` of the enclosing module. This is also synchronized across threads, so that code can safely rely upon this ordering of effects, such that all `__init__` will have run, in dependency ordering, before the `using` result is completed. They may run concurrently with other `__init__` methods which are not dependencies however, so be careful when accessing any shared state outside the current module to use locks when needed.

Two typical uses of `__init__` are calling runtime initialization functions of external C libraries and initializing global constants that involve pointers returned by external libraries. For example, suppose that we are calling a C library `libfoo` that requires us to call a `foo_init()` initialization function at runtime. Suppose that we also want to define a global constant `foo_data_ptr` that holds the return value of a `void *foo_data()` function defined by `libfoo` – this constant must be initialized at runtime (not at compile time) because the pointer address will change from run to run. You could accomplish this by defining the following `__init__` function in your module:

```
const foo_data_ptr = Ref{Ptr{Cvoid}}(0)
function __init__()
    ccall((:foo_init, :libfoo), Cvoid, ())
    foo_data_ptr[] = ccall((:foo_data, :libfoo), Ptr{Cvoid}, ())
    nothing
end
```

Notice that it is perfectly possible to define a global inside a function like `__init__`; this is one of the advantages of using a dynamic language. But by making it a constant at global scope, we can ensure that the type is known to the compiler and allow it to generate better optimized code. Obviously, any other globals in your module that depends on `foo_data_ptr` would also have to be initialized in `__init__`.

Constants involving most Julia objects that are not produced by `ccall` do not need to be placed in `__init__`: their definitions can be precompiled and loaded from the cached module image. This includes complicated heap-allocated objects like arrays. However, any routine that returns a raw pointer value must be called at runtime for precompilation to work (`Ptr` objects will turn into null pointers unless they are hidden inside an `isbits` object). This includes the return values of the Julia functions `@cfunction` and `pointer`.

When using precompilation, it is important to keep a clear sense of the distinction between the compilation phase and the execution phase. In this mode, it will often be much more clearly apparent that Julia is a compiler which allows execution of arbitrary Julia code, not a standalone interpreter that also generates compiled code.

Other known potential failure scenarios include:

1. Global counters (for example, for attempting to uniquely identify objects). Consider the following code snippet:

   ```
   mutable struct UniquedById
       myid::Int
       let counter = 0
           UniquedById() = new(counter += 1)
       end
   end
   ```

   while the intent of this code was to give every instance a unique id, the counter value is recorded at the end of compilation. All subsequent usages of this incrementally compiled module will start from that same counter value.

   Note that `objectid` (which works by hashing the memory pointer) has similar issues (see notes on `Dict` usage below).

   One alternative is to use a macro to capture `@__MODULE__` and store it alone with the current `counter` value, however, it may be better to redesign the code to not depend on this global state.

2. Associative collections (such as `Dict` and `Set`) need to be re-hashed in `__init__`. (In the future, a mechanism may be provided to register an initializer function.)

3. Depending on compile-time side-effects persisting through load-time. Example include: modifying arrays or other variables in other Julia modules; maintaining handles to open files or devices; storing pointers to other system resources (including memory);

4. Creating accidental "copies" of global state from another module, by referencing it directly instead of via its lookup path. For example, (in global scope):

```
#mystdout = Base.stdout #= will not work correctly, since this will copy Base.stdout into
↪  this module =#
# instead use accessor functions:
getstdout() = Base.stdout #= best option =#
# or move the assignment into the runtime:
__init__() = global mystdout = Base.stdout #= also works =#
```

Several additional restrictions are placed on the operations that can be done while precompiling code to help the user avoid other wrong-behavior situations:

1. Calling `eval` to cause a side-effect in another module. This will also cause a warning to be emitted when the incremental precompile flag is set.

2. `global const` statements from local scope after `__init__()` has been started (see issue #12010 for plans to add an error for this)

3. Replacing a module is a runtime error while doing an incremental precompile.

A few other points to be aware of:

1. No code reload / cache invalidation is performed after changes are made to the source files themselves, (including by `Pkg.update`), and no cleanup is done after `Pkg.rm`

2. The memory sharing behavior of a reshaped array is disregarded by precompilation (each view gets its own copy)

3. Expecting the filesystem to be unchanged between compile-time and runtime e.g. `@__FILE__/source_path()` to find resources at runtime, or the BinDeps `@checked_lib` macro. Sometimes this is unavoidable. However, when possible, it can be good practice to copy resources into the module at compile-time so they won't need to be found at runtime.

4. `WeakRef` objects and finalizers are not currently handled properly by the serializer (this will be fixed in an upcoming release).

5. It is usually best to avoid capturing references to instances of internal metadata objects such as `Method`, `MethodInstance`, `MethodTable`, `TypeMapLevel`, `TypeMapEntry` and fields of those objects, as this can confuse the serializer and may not lead to the outcome you desire. It is not necessarily an error to do this, but you simply need to be prepared that the system will try to copy some of these and to create a single unique instance of others.

It is sometimes helpful during module development to turn off incremental precompilation. The command line flag `--compiled-modules={yes|no|existing}` enables you to toggle module precompilation on and off. When Julia is started with `--compiled-modules=no` the serialized modules in the compile cache are ignored when loading modules and module dependencies. In some cases, you may want to load existing precompiled modules, but not create new ones. This can be done by starting Julia with `--compiled-modules=existing`. More fine-grained control is available with `--pkgimages={yes|no|existing}`, which only affects native-code storage during precompilation. `Base.compilecache` can still be called manually. The state of this command line flag is passed to `Pkg.build` to disable automatic precompilation triggering when installing, updating, and explicitly building packages.

You can also debug some precompilation failures with environment variables. Setting `JULIA_VERBOSE_LINKING=true` may help resolve failures in linking shared libraries of compiled native code. See the **Developer Documentation** part of the Julia manual, where you will find further details in the section documenting Julia's internals under "Package Images".

Chapter 18

Documentation

18.1 Accessing Documentation

Documentation can be accessed at the REPL or in IJulia by typing `?` followed by the name of a function or macro, and pressing `Enter`. For example,

```
?cos
?@time
?r""
```

will show documentation for the relevant function, macro or string macro respectively. Most Julia environments provide a way to access documentation directly:

- VS Code shows documentation when you hover over a function name. You can also use the Julia panel in the sidebar to search for documentation.
- In Pluto, open the "Live Docs" panel on the bottom right.
- In Juno using `Ctrl-J, Ctrl-D` will show the documentation for the object under the cursor.

`Docs.hasdoc(module, name)::Bool` tells whether a name has a docstring. `Docs.undocumented_names(module; all)` returns the undocumented names in a module.

18.2 Writing Documentation

Julia enables package developers and users to document functions, types and other objects easily via a built-in documentation system.

The basic syntax is simple: any string appearing just before an object (function, macro, type or instance) will be interpreted as documenting it (these are called *docstrings*). Note that no blank lines or comments may intervene between a docstring and the documented object. Here is a basic example:

```
"Tell whether there are too many foo items in the array."
foo(xs::Array) = ...
```

Reminder

Any empty lines between the docstring and the object being documented detach the former from the latter, making the docstring ineffective.

Documentation is interpreted as Markdown, so you can use indentation and code fences to delimit code examples from text. Technically, any object can be associated with any other as metadata; Markdown happens to be the default, but one can construct other string macros and pass them to the `@doc` macro just as well.

Note

Markdown support is implemented in the `Markdown` standard library and for a full list of supported syntax see the documentation.

Here is a more complex example, still using Markdown:

```
"""
    bar(x[, y])

Compute the Bar index between `x` and `y`.

If `y` is unspecified, compute the Bar index between all pairs of columns of `x`.

# Examples
```julia-repl
julia> bar([1, 2], [1, 2])
1
```
"""
function bar(x, y) ...
```

As in the example above, we recommend following some simple conventions when writing documentation:

1. Always show the signature of a function at the top of the documentation, with a four-space indent so that it is printed as Julia code.

   This can be identical to the signature present in the Julia code (like `mean(x::AbstractArray)`), or a simplified form. Optional arguments should be represented with their default values (i.e. `f(x, y=1)`) when possible, following the actual Julia syntax. Optional arguments which do not have a default value should be put in brackets (i.e. `f(x[, y])` and `f(x[, y[, z]])`). An alternative solution is to use several lines: one without optional arguments, the other(s) with them. This solution can also be used to document several related methods of a given function. When a function accepts many keyword arguments, only include a `<keyword arguments>` placeholder in the signature (i.e. `f(x; <keyword arguments>)`), and give the complete list under an `# Arguments` section (see point 4 below).

   Use this style to document the return type or give the return value a name:

   ```
   # Naming the return value or its type is not necessary (this is the most common case)
   """
      sum(itr; [init])
   ```

```
...
"""

# The return type is easily documented and critical to the semantics of this function
"""
   vec(x::AbstractArray)::AbstractVector

...
"""

# Naming and/or destructuring the return value clarifies the semantics of this function
"""
   splitdir(path::AbstractString) -> (dir::AbstractString, file::AbstractString)
...
"""
```

When included, a return type should be written after the signature, separated by `::`, while a named return value should be separated by `->`, with a space on both sides. Return types and return values should be valid Julia expressions when possible. Macro docstring signatures that annotate return types or return values should use parentheses to clarify where the macro arguments end and return type or return value begins.

2. Include a single one-line sentence describing what the function does or what the object represents after the simplified signature block. If needed, provide more details in a second paragraph, after a blank line.

   The one-line sentence should use the imperative form ("Do this", "Return that") instead of the third person (do not write "Returns the length...") when documenting functions. It should end with a period. If the meaning of a function cannot be summarized easily, splitting it into separate composable parts could be beneficial (this should not be taken as an absolute requirement for every single case though).

3. Do not repeat yourself.

   Since the function name is given by the signature, there is no need to start the documentation with "The function `bar`...": go straight to the point. Similarly, if the signature specifies the types of the arguments, mentioning them in the description is redundant.

4. Only provide an argument list when really necessary.

   For simple functions, it is often clearer to mention the role of the arguments directly in the description of the function's purpose. An argument list would only repeat information already provided elsewhere. However, providing an argument list can be a good idea for complex functions with many arguments (in particular keyword arguments). In that case, insert it after the general description of the function, under an `# Arguments` header, with one `-` bullet for each argument. The list should mention the types and default values (if any) of the arguments:

```
"""
...
# Arguments
- `n::Integer`: the number of elements to compute.
- `dim::Integer=1`: the dimensions along which to perform the computation.
...
"""
```

5. Provide hints to related functions.

   Sometimes there are functions of related functionality. To increase discoverability please provide a short list of these in a `See also` paragraph.

   ```
   See also [`bar!`](@ref), [`baz`](@ref), [`baaz`](@ref).
   ```

6. Include any code examples in an `# Examples` section.

   Examples should, whenever possible, be written as *doctests*. A *doctest* is a fenced code block (see Code blocks) starting with ```` ```jldoctest ```` and contains any number of `julia>` prompts together with inputs and expected outputs that mimic the Julia REPL.

   Note

   Doctests are enabled by `Documenter.jl`. For more detailed documentation see Documenter's manual.

   For example in the following docstring a variable a is defined and the expected result, as printed in a Julia REPL, appears afterwards:

   ````
   """
   Some nice documentation here.

   # Examples
   ```jldoctest
   julia> a = [1 2; 3 4]
   2×2 Matrix{Int64}:
    1  2
    3  4
   ```
   """
   ````

   Warning

   Calling `rand` and other RNG-related functions should be avoided in doctests since they will not produce consistent outputs during different Julia sessions. If you would like to show some random number generation related functionality, one option is to explicitly construct and seed your own RNG object (see `Random`) and pass it to the functions you are doctesting.

   Operating system word size (`Int32` or `Int64`) as well as path separator differences (`/` or `\`) will also affect the reproducibility of some doctests.

   Note that whitespace in your doctest is significant! The doctest will fail if you misalign the output of pretty-printing an array, for example.

   You can then run `make -C doc doctest=true` to run all the doctests in the Julia Manual and API documentation, which will ensure that your example works.

   To indicate that the output result is truncated, you may write `[...]` at the line where checking should stop. This is useful to hide a stacktrace (which contains non-permanent references to lines of julia code) when the doctest shows that an exception is thrown, for example:

   ````
   ```jldoctest
   julia> div(1, 0)
   ````
   ````

```
ERROR: DivideError: integer division error
[...]
```
```

Examples that are untestable should be written within fenced code blocks starting with ```` ```julia ```` so that they are highlighted correctly in the generated documentation.

**Tip**

Wherever possible examples should be **self-contained** and **runnable** so that readers are able to try them out without having to include any dependencies.

7. Use backticks to identify code and equations.

   Julia identifiers and code excerpts should always appear between backticks `` ` `` to enable highlighting. Equations in the LaTeX syntax can be inserted between double backticks ``` `` ```. Use Unicode characters rather than their LaTeX escape sequence, i.e. ```` ``α = 1`` ```` rather than ```` ``\\alpha = 1`` ````.

8. Place the starting and ending `"""` characters on lines by themselves.

   That is, write:

   ```
   """
   ...

   ...
   """
   f(x, y) = ...
   ```

   rather than:

   ```
   """...

   ..."""
   f(x, y) = ...
   ```

   This makes it clearer where docstrings start and end.

9. Respect the line length limit used in the surrounding code.

   Docstrings are edited using the same tools as code. Therefore, the same conventions should apply. It is recommended that lines are at most 92 characters wide.

10. Provide information allowing custom types to implement the function in an `# Implementation` section. These implementation details are intended for developers rather than users, explaining e.g. which functions should be overridden and which functions automatically use appropriate fallbacks. Such details are best kept separate from the main description of the function's behavior.

11. For long docstrings, consider splitting the documentation with an `# Extended help` header. The typical help-mode will show only the material above the header; you can access the full help by adding a '?' at the beginning of the expression (i.e., "??foo" rather than "?foo").
   ```

18.3 Functions & Methods

Functions in Julia may have multiple implementations, known as methods. While it's good practice for generic functions to have a single purpose, Julia allows methods to be documented individually if necessary. In general, only the most generic method should be documented, or even the function itself (i.e. the object created without any methods by `function bar end`). Specific methods should only be documented if their behaviour differs from the more generic ones. In any case, they should not repeat the information provided elsewhere. For example:

```
"""
    *(x, y, z...)

Multiplication operator. `x * y * z *...` calls this function with multiple
arguments, i.e. `*(x, y, z...)`.
"""
function *(x, y, z...)
    # ... [implementation sold separately] ...
end

"""
    *(x::AbstractString, y::AbstractString, z::AbstractString...)

When applied to strings, concatenates them.
"""
function *(x::AbstractString, y::AbstractString, z::AbstractString...)
    # ... [insert secret sauce here] ...
end

help?> *
search: * .*

  *(x, y, z...)

  Multiplication operator. x * y * z *... calls this function with multiple
  arguments, i.e. *(x,y,z...).

  *(x::AbstractString, y::AbstractString, z::AbstractString...)

  When applied to strings, concatenates them.
```

When retrieving documentation for a generic function, the metadata for each method is concatenated with the `catdoc` function, which can of course be overridden for custom types.

18.4 Advanced Usage

The `@doc` macro associates its first argument with its second in a per-module dictionary called `META`.

To make it easier to write documentation, the parser treats the macro name `@doc` specially: if a call to `@doc` has one argument, but another expression appears after a single line break, then that additional expression is added as an argument to the macro. Therefore the following syntax is parsed as a 2-argument call to `@doc`:

```
@doc raw"""
...
"""
f(x) = x
```

This makes it possible to use expressions other than normal string literals (such as the `raw""` string macro) as a docstring.

When used for retrieving documentation, the `@doc` macro (or equally, the `doc` function) will search all `META` dictionaries for metadata relevant to the given object and return it. The returned object (some Markdown content, for example) will by default display itself intelligently. This design also makes it easy to use the doc system in a programmatic way; for example, to re-use documentation between different versions of a function:

```
@doc "..." foo!
@doc (@doc foo!) foo
```

Julia 1.11

In Julia 1.11 and newer, retrieving documentation with the `@doc` macro requires that the REPL stdlib is loaded.

Or for use with Julia's metaprogramming functionality:

```
for (f, op) in ((:add, :+), (:subtract, :-), (:multiply, :*), (:divide, :/))
    @eval begin
        $f(a, b) = $op(a, b)
    end
end
@doc "`add(a, b)` adds `a` and `b` together" add
@doc "`subtract(a, b)` subtracts `b` from `a`" subtract
```

Documentation in non-toplevel blocks, such as `begin`, `if`, `for`, `let`, and inner constructors, should be added to the documentation system via `@doc` as well. For example:

```
if condition()
    @doc "..."
    f(x) = x
end
```

will add documentation to `f(x)` when `condition()` is `true`. Note that even if `f(x)` goes out of scope at the end of a block, its documentation will remain.

It is possible to make use of metaprogramming to assist in the creation of documentation. When using string-interpolation within the docstring you will need to use an extra `$` as shown with `$($name)`:

```
for func in (:day, :dayofmonth)
    name = string(func)
    @eval begin
```

```
        @doc """
            $($name)(dt::TimeType) -> Int64

        The day of month of a `Date` or `DateTime` as an `Int64`.
        """ $func(dt::Dates.TimeType)
    end
end
```

Dynamic documentation

Sometimes the appropriate documentation for an instance of a type depends on the field values of that instance, rather than just on the type itself. In these cases, you can add a method to `Docs.getdoc` for your custom type that returns the documentation on a per-instance basis. For instance,

```
struct MyType
    value::Int
end

Docs.getdoc(t::MyType) = "Documentation for MyType with value $(t.value)"

x = MyType(1)
y = MyType(2)
```

?x will display "Documentation for MyType with value 1" while ?y will display "Documentation for MyType with value 2".

18.5 Syntax Guide

This guide provides a comprehensive overview of how to attach documentation to all Julia syntax constructs for which providing documentation is possible.

In the following examples `"..."` is used to illustrate an arbitrary docstring.

$ and \ characters

The $ and \ characters are still parsed as string interpolation or start of an escape sequence in docstrings too. The `raw""` string macro together with the `@doc` macro can be used to avoid having to escape them. This is handy when the docstrings include LaTeX or Julia source code examples containing interpolation:

```
@doc raw"""
```math
\LaTeX
```
"""
function f end
```

Functions and Methods

```
"..."
function f end
```

```
"..."
f
```

Adds docstring `"..."` to the function `f`. The first version is the preferred syntax, however both are equivalent.

```
"..."
f(x) = x

"..."
function f(x)
    return x
end

"..."
f(x)
```

Adds docstring `"..."` to the method `f(::Any)`.

```
"..."
f(x, y = 1) = x + y
```

Adds docstring `"..."` to two Methods, namely `f(::Any)` and `f(::Any, ::Any)`.

Macros

```
"..."
macro m(x) end
```

Adds docstring `"..."` to the `@m(::Any)` macro definition.

```
"..."
:(@m1)

"..."
macro m2 end
```

Adds docstring `"..."` to the macros named `@m1` and `@m2`.

Types

```
"..."
abstract type T1 end

"..."
mutable struct T2
    ...
end
```

```
"..."
struct T3
    ...
end
```

Adds the docstring `"..."` to types T1, T2, and T3.

```
"..."
T1

"..."
T2

"..."
T3
```

Adds the docstring `"..."` to types T1, T2, and T3. The previous version is the preferred syntax, however both are equivalent.

```
"..."
struct T
    "x"
    x
    "y"
    y

    @doc "Inner constructor"
    function T()
        new(...)
    end
end
```

Adds docstring `"..."` to type T, `"x"` to field `T.x`, `"y"` to field `T.y`, and `"Inner constructor"` to the inner constructor `T()`. Also applicable to `mutable struct` types.

Modules

```
"..."
module M end

module M

"..."
M

end
```

Adds docstring `"..."` to the `Module` M. Adding the docstring above the `Module` is the preferred syntax, however both are equivalent.

The module docstring is evaluated *inside* the scope of the module, allowing access to all the symbols defined in and imported into the module:

```
"The magic number is $(MAGIC)."
module DocStringEval
const MAGIC = 42
end
```

Documenting a `baremodule` by placing a docstring above the expression automatically imports `@doc` into the module. These imports must be done manually when the module expression is not documented:

```
"..."
baremodule M
# ...
end

baremodule M

import Base: @doc

"..."
f(x) = x

end
```

Global Variables

```
"..."
const a = 1

"..."
b = 2

"..."
global c = 3
```

Adds docstring `"..."` to the `Binding`s `a`, `b`, and `c`.

`Binding`s are used to store a reference to a particular `Symbol` in a `Module` without storing the referenced value itself.

Note

When a `const` definition is only used to define an alias of another definition, such as is the case with the function `div` and its alias `÷` in `Base`, do not document the alias and instead document the actual function.

If the alias is documented and not the real definition then the docsystem (`?` mode) will not return the docstring attached to the alias when the real definition is searched for.

For example you should write

```
"..."
f(x) = x + 1
const alias = f
```

rather than

```
f(x) = x + 1
"..."
const alias = f
```

```
"..."
sym
```

Adds docstring `"..."` to the value associated with `sym`. However, it is preferred that `sym` is documented where it is defined.

Multiple Objects

```
"..."
a, b
```

Adds docstring `"..."` to `a` and `b` each of which should be a documentable expression. This syntax is equivalent to

```
"..."
a

"..."
b
```

Any number of expressions many be documented together in this way. This syntax can be useful when two functions are related, such as non-mutating and mutating versions `f` and `f!`.

Macro-generated code

```
"..."
@m expression
```

Adds docstring `"..."` to the expression generated by expanding `@m expression`. This allows for expressions decorated with `@inline`, `@noinline`, `@generated`, or any other macro to be documented in the same way as undecorated expressions.

Macro authors should take note that only macros that generate a single expression will automatically support docstrings. If a macro returns a block containing multiple subexpressions then the subexpression that should be documented must be marked using the `@__doc__` macro.

The `@enum` macro makes use of `@__doc__` to allow for documenting `Enum`s. Examining its definition should serve as an example of how to use `@__doc__` correctly.

`Core.@__doc__` – Macro.

```
@__doc__(ex)
```

Low-level macro used to mark expressions returned by a macro that should be documented. If more than one expression is marked then the same docstring is applied to each expression.

```
macro example(f)
    quote
        $(f)() = 0
        @__doc__ $(f)(x) = 1
        $(f)(x, y) = 2
    end |> esc
end
```

`@__doc__` has no effect when a macro that uses it is not documented.

Julia 1.12

This section documents a very subtle corner case that is only relevant to macros which themselves both define other macros and then attempt to use them within the same expansion. Such macros were impossible to write prior to Julia 1.12 and are still quite rare. If you are not writing such a macro, you may ignore this note.

In versions prior to Julia 1.12, macroexpansion would recursively expand through `Expr(:toplevel)` blocks. This behavior was changed in Julia 1.12 to allow macros to recursively define other macros and use them in the same returned expression. However, to preserve backwards compatibility with existing uses of `@__doc__`, the doc system will still expand through `Expr(:toplevel)` blocks when looking for `@__doc__` markers. As a result, macro-defining-macros will have an observable behavior difference when annotated with a docstring:

```
julia> macro macroception()
    Expr(:toplevel, :(macro foo() 1 end), :(@foo))
end

julia> @macroception
1

julia> "Docstring" @macroception
ERROR: LoadError: UndefVarError: `@foo` not defined in `Main`
```

The supported workaround is to manually expand the `@__doc__` macro in the defining macro, which the docsystem will recognize and suppress the recursive expansion:

```
julia> macro macroception()
    Expr(:toplevel,
        macroexpand(__module__, :(@__doc__ macro foo() 1 end); recursive=false),
        :(@foo))
end

julia> @macroception
1

julia> "Docstring" @macroception
1
```

source

Chapter 19

Metaprogramming

The strongest legacy of Lisp in the Julia language is its metaprogramming support. Like Lisp, Julia represents its own code as a data structure of the language itself. Since code is represented by objects that can be created and manipulated from within the language, it is possible for a program to transform and generate its own code. This allows sophisticated code generation without extra build steps, and also allows true Lisp-style macros operating at the level of abstract syntax trees. In contrast, preprocessor "macro" systems, like that of C and C++, perform textual manipulation and substitution before any actual parsing or interpretation occurs. Because all data types and code in Julia are represented by Julia data structures, powerful reflection capabilities are available to explore the internals of a program and its types just like any other data.

Warning

Metaprogramming is a powerful tool, but it introduces complexity that can make code more difficult to understand. For example, it can be surprisingly hard to get scope rules correct. Metaprogramming should typically be used only when other approaches such as higher order functions and closures cannot be applied.

`eval` and defining new macros should be typically used as a last resort. It is almost never a good idea to use `Meta.parse` or convert an arbitrary string into Julia code. For manipulating Julia code, use the `Expr` data structure directly to avoid the complexity of how Julia syntax is parsed.

The best uses of metaprogramming often implement most of their functionality in runtime helper functions, striving to minimize the amount of code they generate.

19.1 Program representation

Every Julia program starts life as a string:

```
julia> prog = "1 + 1"
"1 + 1"
```

What happens next?

The next step is to parse each string into an object called an expression, represented by the Julia type `Expr`:

```
julia> ex1 = Meta.parse(prog)
:(1 + 1)

julia> typeof(ex1)
Expr
```

`Expr` objects contain two parts:

- a `Symbol` identifying the kind of expression. A symbol is an interned string identifier (more discussion below).

```
julia> ex1.head
:call
```

- the expression arguments, which may be symbols, other expressions, or literal values:

```
julia> ex1.args
3-element Vector{Any}:
  :+
 1
 1
```

Expressions may also be constructed directly in prefix notation:

```
julia> ex2 = Expr(:call, :+, 1, 1)
:(1 + 1)
```

The two expressions constructed above – by parsing and by direct construction – are equivalent:

```
julia> ex1 == ex2
true
```

The key point here is that Julia code is internally represented as a data structure that is accessible from the language itself.

The `dump` function provides indented and annotated display of `Expr` objects:

```
julia> dump(ex2)
Expr
  head: Symbol call
  args: Array{Any}((3,))
    1: Symbol +
    2: Int64 1
    3: Int64 1
```

`Expr` objects may also be nested:

```
julia> ex3 = Meta.parse("(4 + 4) / 2")
:((4 + 4) / 2)
```

Another way to view expressions is with `Meta.show_sexpr`, which displays the S-expression form of a given `Expr`, which may look very familiar to users of Lisp. Here's an example illustrating the display on a nested `Expr`:

```
julia> Meta.show_sexpr(ex3)
(:call, :/, (:call, :+, 4, 4), 2)
```

Symbols

The `:` character has two syntactic purposes in Julia. The first form creates a `Symbol`, an interned string used as one building-block of expressions, from valid identifiers:

```
julia> s = :foo
:foo

julia> typeof(s)
Symbol
```

The `Symbol` constructor takes any number of arguments and creates a new symbol by concatenating their string representations together:

```
julia> :foo === Symbol("foo")
true

julia> Symbol("1foo") # `:1foo` would not work, as `1foo` is not a valid identifier
Symbol("1foo")

julia> Symbol("func",10)
:func10

julia> Symbol(:var,'_',"sym")
:var_sym
```

In the context of an expression, symbols are used to indicate access to variables; when an expression is evaluated, a symbol is replaced with the value bound to that symbol in the appropriate scope.

Sometimes extra parentheses around the argument to `:` are needed to avoid ambiguity in parsing:

```
julia> :(:)
:(:)

julia> :(::)
:(::)
```

19.2 Expressions and evaluation

Quoting

The second syntactic purpose of the `:` character is to create expression objects without using the explicit `Expr` constructor. This is referred to as *quoting*. The `:` character, followed by paired parentheses around a single statement of Julia code, produces an `Expr` object based on the enclosed code. Here is an example of the short form used to quote an arithmetic expression:

```
julia> ex = :(a+b*c+1)
:(a + b * c + 1)

julia> typeof(ex)
Expr
```

(to view the structure of this expression, try `ex.head` and `ex.args`, or use `dump` as above or `Meta.@dump`)

Note that equivalent expressions may be constructed using `Meta.parse` or the direct `Expr` form:

```
julia>      :(a + b*c + 1)       ==
       Meta.parse("a + b*c + 1") ==
       Expr(:call, :+, :a, Expr(:call, :*, :b, :c), 1)
true
```

Expressions provided by the parser generally only have symbols, other expressions, and literal values as their args, whereas expressions constructed by Julia code can have arbitrary run-time values without literal forms as args. In this specific example, `+` and `a` are symbols, `*(b,c)` is a subexpression, and `1` is a literal 64-bit signed integer.

There is a second syntactic form of quoting for multiple expressions: blocks of code enclosed in `quote ... end`.

```
julia> ex = quote
           x = 1
           y = 2
           x + y
       end
quote
    #= none:2 =#
    x = 1
    #= none:3 =#
    y = 2
    #= none:4 =#
    x + y
end

julia> typeof(ex)
Expr
```

Interpolation

Direct construction of `Expr` objects with value arguments is powerful, but `Expr` constructors can be tedious compared to "normal" Julia syntax. As an alternative, Julia allows *interpolation* of literals or expressions into quoted expressions. Interpolation is indicated by a prefix `$`.

In this example, the value of variable a is interpolated:

```
julia> a = 1;

julia> ex = :($a + b)
:(1 + b)
```

Interpolating into an unquoted expression is not supported and will cause a compile-time error:

```
julia> $a + b
ERROR: syntax: "$" expression outside quote
```

In this example, the tuple `(1,2,3)` is interpolated as an expression into a conditional test:

```
julia> ex = :(a in $:((1,2,3)) )
:(a in (1, 2, 3))
```

The use of `$` for expression interpolation is intentionally reminiscent of string interpolation and command interpolation. Expression interpolation allows convenient, readable programmatic construction of complex Julia expressions.

Splatting interpolation

Notice that the `$` interpolation syntax allows inserting only a single expression into an enclosing expression. Occasionally, you have an array of expressions and need them all to become arguments of the surrounding expression. This can be done with the syntax `$(xs...)`. For example, the following code generates a function call where the number of arguments is determined programmatically:

```
julia> args = [:x, :y, :z];

julia> :(f(1, $(args...)))
:(f(1, x, y, z))
```

Nested quote

Naturally, it is possible for quote expressions to contain other quote expressions. Understanding how interpolation works in these cases can be a bit tricky. Consider this example:

```
julia> x = :(1 + 2);

julia> e = quote quote $x end end
quote
    #= none:1 =#
    $(Expr(:quote, quote
```

```
    #= none:1 =#
    $(Expr(:$, :x))
end))
end
```

Notice that the result contains $x, which means that x has not been evaluated yet. In other words, the $ expression "belongs to" the inner quote expression, and so its argument is only evaluated when the inner quote expression is:

```
julia> eval(e)
quote
    #= none:1 =#
    1 + 2
end
```

However, the outer `quote` expression is able to interpolate values inside the `$` in the inner quote. This is done with multiple $s:

```
julia> e = quote quote $$x end end
quote
    #= none:1 =#
    $(Expr(:quote, quote
    #= none:1 =#
    $(Expr(:$, :(1 + 2)))
end))
end
```

Notice that `(1 + 2)` now appears in the result instead of the symbol `x`. Evaluating this expression yields an interpolated 3:

```
julia> eval(e)
quote
    #= none:1 =#
    3
end
```

The intuition behind this behavior is that x is evaluated once for each `$`: one `$` works similarly to `eval(:x)`, giving x's value, while two $s do the equivalent of `eval(eval(:x))`.

QuoteNode

The usual representation of a `quote` form in an AST is an `Expr` with head `:quote`:

```
julia> dump(Meta.parse(":(1+2)"))
Expr
  head: Symbol quote
  args: Array{Any}((1,))
    1: Expr
      head: Symbol call
```

```
    args: Array{Any}((3,))
      1: Symbol +
      2: Int64 1
      3: Int64 2
```

As we have seen, such expressions support interpolation with `$`. However, in some situations it is necessary to quote code *without* performing interpolation. This kind of quoting does not yet have syntax, but is represented internally as an object of type `QuoteNode`:

```
julia> eval(Meta.quot(Expr(:$, :(1+2))))
3

julia> eval(QuoteNode(Expr(:$, :(1+2))))
:($(Expr(:$, :(1 + 2))))
```

The parser yields `QuoteNode`s for simple quoted items like symbols:

```
julia> dump(Meta.parse(":x"))
QuoteNode
  value: Symbol x
```

`QuoteNode` can also be used for certain advanced metaprogramming tasks.

Note that while it does not support `$`, it also does not prevent it, nor does it preserve the identity of the wrapped object:

```
julia> b = 2; eval(Expr(:quote, QuoteNode(Expr(:$, :b))))
:($(QuoteNode(2)))
```

Evaluating expressions

Given an expression object, one can cause Julia to evaluate (execute) it at global scope using `eval`:

```
julia> ex1 = :(1 + 2)
:(1 + 2)

julia> eval(ex1)
3

julia> ex = :(a + b)
:(a + b)

julia> eval(ex)
ERROR: UndefVarError: `b` not defined in `Main`
[...]

julia> a = 1; b = 2;

julia> eval(ex)
3
```

Every module has its own `eval` function that evaluates expressions in its global scope. Expressions passed to `eval` are not limited to returning values – they can also have side-effects that alter the state of the enclosing module's environment:

```
julia> ex = :(x = 1)
:(x = 1)

julia> x
ERROR: UndefVarError: `x` not defined in `Main`

julia> eval(ex)
1

julia> x
1
```

Here, the evaluation of an expression object causes a value to be assigned to the global variable `x`.

Since expressions are just `Expr` objects which can be constructed programmatically and then evaluated, it is possible to dynamically generate arbitrary code which can then be run using `eval`. Here is a simple example:

```
julia> a = 1;

julia> ex = Expr(:call, :+, a, :b)
:(1 + b)

julia> a = 0; b = 2;

julia> eval(ex)
3
```

The value of `a` is used to construct the expression `ex` which applies the `+` function to the value 1 and the variable `b`. Note the important distinction between the way `a` and `b` are used:

- The value of the *variable* `a` at expression construction time is used as an immediate value in the expression. Thus, the value of `a` when the expression is evaluated no longer matters: the value in the expression is already 1, independent of whatever the value of `a` might be.
- On the other hand, the *symbol* `:b` is used in the expression construction, so the value of the variable `b` at that time is irrelevant – `:b` is just a symbol and the variable `b` need not even be defined. At expression evaluation time, however, the value of the symbol `:b` is resolved by looking up the value of the variable `b`.

Functions on Expressions

As hinted above, one extremely useful feature of Julia is the capability to generate and manipulate Julia code within Julia itself. We have already seen one example of a function returning `Expr` objects: the `Meta.parse` function, which takes a string of Julia code and returns the corresponding `Expr`. A function can also take one or more `Expr` objects as arguments, and return another `Expr`. Here is a simple, motivating example:

```
julia> function math_expr(op, op1, op2)
           expr = Expr(:call, op, op1, op2)
           return expr
       end
math_expr (generic function with 1 method)

julia>  ex = math_expr(:+, 1, Expr(:call, :*, 4, 5))
:(1 + 4 * 5)

julia> eval(ex)
21
```

As another example, here is a function that doubles any numeric argument, but leaves expressions alone:

```
julia> function make_expr2(op, opr1, opr2)
           opr1f, opr2f = map(x -> isa(x, Number) ? 2*x : x, (opr1, opr2))
           retexpr = Expr(:call, op, opr1f, opr2f)
           return retexpr
       end
make_expr2 (generic function with 1 method)

julia> make_expr2(:+, 1, 2)
:(2 + 4)

julia> ex = make_expr2(:+, 1, Expr(:call, :*, 5, 8))
:(2 + 5 * 8)

julia> eval(ex)
42
```

19.3 Macros

Macros provide a mechanism to include generated code in the final body of a program. A macro maps a tuple of arguments to a returned *expression*, and the resulting expression is compiled directly rather than requiring a runtime `eval` call. Macro arguments may include expressions, literal values, and symbols.

Basics

Here is an extraordinarily simple macro:

```
julia> macro sayhello()
           return :( println("Hello, world!") )
       end
@sayhello (macro with 1 method)
```

Macros have a dedicated character in Julia's syntax: the `@` (at-sign), followed by the unique name declared in a `macro NAME ... end` block. In this example, the compiler will replace all instances of `@sayhello` with:

```
:( println("Hello, world!") )
```

When `@sayhello` is entered in the REPL, the expression executes immediately, thus we only see the evaluation result:

```
julia> @sayhello()
Hello, world!
```

Now, consider a slightly more complex macro:

```
julia> macro sayhello(name)
           return :( println("Hello, ", $name) )
       end
@sayhello (macro with 1 method)
```

This macro takes one argument: `name`. When `@sayhello` is encountered, the quoted expression is *expanded* to interpolate the value of the argument into the final expression:

```
julia> @sayhello("human")
Hello, human
```

We can view the quoted return expression using the function `macroexpand` (**important note:** this is an extremely useful tool for debugging macros):

```
julia> ex = macroexpand(Main, :(@sayhello("human")) )
:(Main.println("Hello, ", "human"))

julia> typeof(ex)
Expr
```

We can see that the `"human"` literal has been interpolated into the expression.

There also exists a macro `@macroexpand` that is perhaps a bit more convenient than the `macroexpand` function:

```
julia> @macroexpand @sayhello "human"
:(println("Hello, ", "human"))
```

Hold up: why macros?

We have already seen a function `f(::Expr...) -> Expr` in a previous section. In fact, `macroexpand` is also such a function. So, why do macros exist?

Macros are necessary because they execute when code is parsed, therefore, macros allow the programmer to generate and include fragments of customized code *before* the full program is run. To illustrate the difference, consider the following example:

```
julia> macro twostep(arg)
           println("I execute at parse time. The argument is: ", arg)
           return :(println("I execute at runtime. The argument is: ", $arg))
       end
```

```
@twostep (macro with 1 method)

julia> ex = macroexpand(Main, :(@twostep :(1, 2, 3)) );
I execute at parse time. The argument is: :((1, 2, 3))
```

The first call to `println` is executed when `macroexpand` is called. The resulting expression contains *only* the second `println`:

```
julia> typeof(ex)
Expr

julia> ex
:(println("I execute at runtime. The argument is: ", $(Expr(:copyast, :($(QuoteNode(:((1, 2,
↪  3)))))))))

julia> eval(ex)
I execute at runtime. The argument is: (1, 2, 3)
```

Macro invocation

Macros are invoked with the following general syntax:

```
@name expr1 expr2 ...
@name(expr1, expr2, ...)
```

Note the distinguishing `@` before the macro name and the lack of commas between the argument expressions in the first form, and the lack of whitespace after `@name` in the second form. The two styles should not be mixed. For example, the following syntax is different from the examples above; it passes the tuple `(expr1, expr2, ...)` as one argument to the macro:

```
@name (expr1, expr2, ...)
```

An alternative way to invoke a macro over an array literal (or comprehension) is to juxtapose both without using parentheses. In this case, the array will be the only expression fed to the macro. The following syntax is equivalent (and different from `@name [a b] * v`):

```
@name[a b] * v
@name([a b]) * v
```

It is important to emphasize that macros receive their arguments as expressions, literals, or symbols. One way to explore macro arguments is to call the `show` function within the macro body:

```
julia> macro showarg(x)
           show(x)
           # ... remainder of macro, returning an expression
       end
@showarg (macro with 1 method)

julia> @showarg(a)
```

```
:a

julia> @showarg(1+1)
:(1 + 1)

julia> @showarg(println("Yo!"))
:(println("Yo!"))

julia> @showarg(1)        # Numeric literal
1

julia> @showarg("Yo!")    # String literal
"Yo!"

julia> @showarg("Yo! $("hello")")    # String with interpolation is an Expr rather than a String
:("Yo! $("hello")")
```

In addition to the given argument list, every macro is passed extra arguments named `__source__` and `__module__`.

The argument `__source__` provides information (in the form of a `LineNumberNode` object) about the parser location of the `@` sign from the macro invocation. This allows macros to include better error diagnostic information, and is commonly used by logging, string-parser macros, and docs, for example, as well as to implement the `@__LINE__`, `@__FILE__`, and `@__DIR__` macros.

The location information can be accessed by referencing `__source__.line` and `__source__.file`:

```
julia> macro __LOCATION__(); return QuoteNode(__source__); end
@__LOCATION__ (macro with 1 method)

julia> dump(
            @__LOCATION__(
       ))
LineNumberNode
  line: Int64 2
  file: Symbol none
```

The argument `__module__` provides information (in the form of a `Module` object) about the expansion context of the macro invocation. This allows macros to look up contextual information, such as existing bindings, or to insert the value as an extra argument to a runtime function call doing self-reflection in the current module.

Building an advanced macro

Here is a simplified definition of Julia's `@assert` macro:

```
julia> macro assert(ex)
           return :( $ex ? nothing : throw(AssertionError($(string(ex)))) )
       end
@assert (macro with 1 method)
```

This macro can be used like this:

```
julia> @assert 1 == 1.0

julia> @assert 1 == 0
ERROR: AssertionError: 1 == 0
```

In place of the written syntax, the macro call is expanded at parse time to its returned result. This is equivalent to writing:

```
1 == 1.0 ? nothing : throw(AssertionError("1 == 1.0"))
1 == 0 ? nothing : throw(AssertionError("1 == 0"))
```

That is, in the first call, the expression `:(1 == 1.0)` is spliced into the test condition slot, while the value of `string(:(1 == 1.0))` is spliced into the assertion message slot. The entire expression, thus constructed, is placed into the syntax tree where the `@assert` macro call occurs. Then at execution time, if the test expression evaluates to true, then `nothing` is returned, whereas if the test is false, an error is raised indicating the asserted expression that was false. Notice that it would not be possible to write this as a function, since only the *value* of the condition is available and it would be impossible to display the expression that computed it in the error message.

The actual definition of `@assert` in Julia Base is more complicated. It allows the user to optionally specify their own error message, instead of just printing the failed expression. Just like in functions with a variable number of arguments (Varargs Functions), this is specified with an ellipses following the last argument:

```
julia> macro assert(ex, msgs...)
           msg_body = isempty(msgs) ? ex : msgs[1]
           msg = string(msg_body)
           return :($ex ? nothing : throw(AssertionError($msg)))
       end
@assert (macro with 1 method)
```

Now `@assert` has two modes of operation, depending upon the number of arguments it receives! If there's only one argument, the tuple of expressions captured by `msgs` will be empty and it will behave the same as the simpler definition above. But now if the user specifies a second argument, it is printed in the message body instead of the failing expression. You can inspect the result of a macro expansion with the aptly named `@macroexpand` macro:

```
julia> @macroexpand @assert a == b
:(if Main.a == Main.b
        Main.nothing
    else
        Main.throw(Main.AssertionError("a == b"))
    end)

julia> @macroexpand @assert a==b "a should equal b!"
:(if Main.a == Main.b
        Main.nothing
    else
        Main.throw(Main.AssertionError("a should equal b!"))
    end)
```

There is yet another case that the actual `@assert` macro handles: what if, in addition to printing "a should equal b," we wanted to print their values? One might naively try to use string interpolation in the custom message, e.g., `@assert a==b "a ($a) should equal b ($b)!"`, but this won't work as expected with the above macro. Can you see why? Recall from string interpolation that an interpolated string is rewritten to a call to `string`. Compare:

```
julia> typeof(:("a should equal b"))
String

julia> typeof(:("a ($a) should equal b ($b)!"))
Expr

julia> dump(:("a ($a) should equal b ($b)!"))
Expr
  head: Symbol string
  args: Array{Any}((5,))
    1: String "a ("
    2: Symbol a
    3: String ") should equal b ("
    4: Symbol b
    5: String ")!"
```

So now instead of getting a plain string in `msg_body`, the macro is receiving a full expression that will need to be evaluated in order to display as expected. This can be spliced directly into the returned expression as an argument to the `string` call; see `error.jl` for the complete implementation.

The `@assert` macro makes great use of splicing into quoted expressions to simplify the manipulation of expressions inside the macro body.

Hygiene

An issue that arises in more complex macros is that of hygiene. In short, macros must ensure that the variables they introduce in their returned expressions do not accidentally clash with existing variables in the surrounding code they expand into. Conversely, the expressions that are passed into a macro as arguments are often *expected* to evaluate in the context of the surrounding code, interacting with and modifying the existing variables. Another concern arises from the fact that a macro may be called in a different module from where it was defined. In this case we need to ensure that all global variables are resolved to the correct module. Julia already has a major advantage over languages with textual macro expansion (like C) in that it only needs to consider the returned expression. All the other variables (such as `msg` in `@assert` above) follow the normal scoping block behavior.

To demonstrate these issues, let us consider writing a `@time` macro that takes an expression as its argument, records the time, evaluates the expression, records the time again, prints the difference between the before and after times, and then has the value of the expression as its final value. The macro might look like this:

```
macro time(ex)
    return quote
        local t0 = time_ns()
        local val = $ex
        local t1 = time_ns()
        println("elapsed time: ", (t1-t0)/1e9, " seconds")
```

```
        val
    end
end
```

Here, we want `t0`, `t1`, and `val` to be private temporary variables, and we want `time_ns` to refer to the `time_ns` function in Julia Base, not to any `time_ns` variable the user might have (the same applies to `println`). Imagine the problems that could occur if the user expression ex also contained assignments to a variable called `t0`, or defined its own `time_ns` variable. We might get errors, or mysteriously incorrect behavior.

Julia's macro expander solves these problems in the following way. First, variables within a macro result are classified as either local or global. A variable is considered local if it is assigned to (and not declared global), declared local, or used as a function argument name. Otherwise, it is considered global. Local variables are then renamed to be unique (using the `gensym` function, which generates new symbols), and global variables are resolved within the macro definition environment. Therefore both of the above concerns are handled; the macro's locals will not conflict with any user variables, and `time_ns` and `println` will refer to the Julia Base definitions.

One problem remains however. Consider the following use of this macro:

```
module MyModule
import Base.@time

time_ns() = ... # compute something

@time time_ns()
end
```

Here the user expression ex is a call to `time_ns`, but not the same `time_ns` function that the macro uses. It clearly refers to `MyModule.time_ns`. Therefore we must arrange for the code in ex to be resolved in the macro call environment. This is done by "escaping" the expression with `esc`:

```
macro time(ex)
    ...
    local val = $(esc(ex))
    ...
end
```

An expression wrapped in this manner is left alone by the macro expander and simply pasted into the output verbatim. Therefore it will be resolved in the macro call environment.

This escaping mechanism can be used to "violate" hygiene when necessary, in order to introduce or manipulate user variables. For example, the following macro sets x to zero in the call environment:

```
julia> macro zerox()
           return esc(:(x = 0))
       end
@zerox (macro with 1 method)

julia> function foo()
           x = 1
```

```
            @zerox
            return x # is zero
        end
foo (generic function with 1 method)

julia> foo()
0
```

This kind of manipulation of variables should be used judiciously, but is occasionally quite handy.

Getting the hygiene rules correct can be a formidable challenge. Before using a macro, you might want to consider whether a function closure would be sufficient. Another useful strategy is to defer as much work as possible to runtime. For example, many macros simply wrap their arguments in a `QuoteNode` or other similar `Expr`. Some examples of this include `@task body` which simply returns `schedule(Task(() -> $body))`, and `@eval expr`, which simply returns `eval(QuoteNode(expr))`.

To demonstrate, we might rewrite the `@time` example above as:

```
macro time(expr)
    return :(timeit(() -> $(esc(expr))))
end
function timeit(f)
    t0 = time_ns()
    val = f()
    t1 = time_ns()
    println("elapsed time: ", (t1-t0)/1e9, " seconds")
    return val
end
```

However, we don't do this for a good reason: wrapping the `expr` in a new scope block (the anonymous function) also slightly changes the meaning of the expression (the scope of any variables in it), while we want `@time` to be usable with minimum impact on the wrapped code.

Macros and dispatch

Macros, just like Julia functions, are generic. This means they can also have multiple method definitions, thanks to multiple dispatch:

```
julia> macro m end
@m (macro with 0 methods)

julia> macro m(args...)
            println("$(length(args)) arguments")
        end
@m (macro with 1 method)

julia> macro m(x,y)
            println("Two arguments")
        end
@m (macro with 2 methods)

julia> @m "asd"
```

```
1 arguments

julia> @m 1 2
Two arguments
```

However one should keep in mind, that macro dispatch is based on the types of AST that are handed to the macro, not the types that the AST evaluates to at runtime:

```
julia> macro m(::Int)
           println("An Integer")
       end
@m (macro with 3 methods)

julia> @m 2
An Integer

julia> x = 2
2

julia> @m x
1 arguments
```

19.4 Code Generation

When a significant amount of repetitive boilerplate code is required, it is common to generate it programmatically to avoid redundancy. In most languages, this requires an extra build step, and a separate program to generate the repetitive code. In Julia, expression interpolation and `eval` allow such code generation to take place in the normal course of program execution. For example, consider the following custom type

```
struct MyNumber
    x::Float64
end
# output
```

for which we want to add a number of methods to. We can do this programmatically in the following loop:

```
for op = (:sin, :cos, :tan, :log, :exp)
    eval(quote
        Base.$op(a::MyNumber) = MyNumber($op(a.x))
    end)
end
# output
```

and we can now use those functions with our custom type:

```
julia> x = MyNumber(π)
MyNumber(3.141592653589793)
```

```
julia> sin(x)
MyNumber(1.2246467991473532e-16)

julia> cos(x)
MyNumber(-1.0)
```

In this manner, Julia acts as its own preprocessor, and allows code generation from inside the language. The above code could be written slightly more tersely using the `:` prefix quoting form:

```
for op = (:sin, :cos, :tan, :log, :exp)
    eval(:(Base.$op(a::MyNumber) = MyNumber($op(a.x))))
end
```

This sort of in-language code generation, however, using the `eval(quote(...))` pattern, is common enough that Julia comes with a macro to abbreviate this pattern:

```
for op = (:sin, :cos, :tan, :log, :exp)
    @eval Base.$op(a::MyNumber) = MyNumber($op(a.x))
end
```

The `@eval` macro rewrites this call to be precisely equivalent to the above longer versions. For longer blocks of generated code, the expression argument given to `@eval` can be a block:

```
@eval begin
    # multiple lines
end
```

19.5 Non-Standard String Literals

Recall from Strings that string literals prefixed by an identifier are called non-standard string literals, and can have different semantics than un-prefixed string literals. For example:

- `r"^\s*(?:#|$)"` produces a regular expression object rather than a string
- `b"DATA\xff\u2200"` is a byte array literal for `[68,65,84,65,255,226,136,128]`.

Perhaps surprisingly, these behaviors are not hard-coded into the Julia parser or compiler. Instead, they are custom behaviors provided by a general mechanism that anyone can use: prefixed string literals are parsed as calls to specially-named macros. For example, the regular expression macro is just the following:

```
macro r_str(p)
    Regex(p)
end
```

That's all. This macro says that the literal contents of the string literal `r"^\s*(?:#|$)"` should be passed to the `@r_str` macro and the result of that expansion should be placed in the syntax tree where the string literal occurs. In other words, the expression `r"^\s*(?:#|$)"` is equivalent to placing the following object directly into the syntax tree:

```
Regex("^\\s*(?:#|\$)")
```

Not only is the string literal form shorter and far more convenient, but it is also more efficient: since the regular expression is compiled and the Regex object is actually created *when the code is compiled*, the compilation occurs only once, rather than every time the code is executed. Consider if the regular expression occurs in a loop:

```
for line = lines
    m = match(r"^\s*(?:#|$)", line)
    if m === nothing
        # non-comment
    else
        # comment
    end
end
```

Since the regular expression `r"^\s*(?:#|$)"` is compiled and inserted into the syntax tree when this code is parsed, the expression is only compiled once instead of each time the loop is executed. In order to accomplish this without macros, one would have to write this loop like this:

```
re = Regex("^\\s*(?:#|\$)")
for line = lines
    m = match(re, line)
    if m === nothing
        # non-comment
    else
        # comment
    end
end
```

Moreover, if the compiler could not determine that the regex object was constant over all loops, certain optimizations might not be possible, making this version still less efficient than the more convenient literal form above. Of course, there are still situations where the non-literal form is more convenient: if one needs to interpolate a variable into the regular expression, one must take this more verbose approach; in cases where the regular expression pattern itself is dynamic, potentially changing upon each loop iteration, a new regular expression object must be constructed on each iteration. In the vast majority of use cases, however, regular expressions are not constructed based on run-time data. In this majority of cases, the ability to write regular expressions as compile-time values is invaluable.

The mechanism for user-defined string literals is deeply, profoundly powerful. Not only are Julia's non-standard literals implemented using it, but the command literal syntax (`` `echo "Hello, $person"` ``) is also implemented using the following innocuous-looking macro:

```
macro cmd(str)
    :(cmd_gen($(shell_parse(str)[1])))
end
```

Of course, a large amount of complexity is hidden in the functions used in this macro definition, but they are just functions, written entirely in Julia. You can read their source and see precisely what they do – and all they do is construct expression objects to be inserted into your program's syntax tree.

Like string literals, command literals can also be prefixed by an identifier to form what are called non-standard command literals. These command literals are parsed as calls to specially-named macros. For example, the syntax `custom`literal`` is parsed as `@custom_cmd "literal"`. Julia itself does not contain any non-standard command literals, but packages can make use of this syntax. Aside from the different syntax and the `_cmd` suffix instead of the `_str` suffix, non-standard command literals behave exactly like non-standard string literals.

In the event that two modules provide non-standard string or command literals with the same name, it is possible to qualify the string or command literal with a module name. For instance, if both `Foo` and `Bar` provide non-standard string literal `@x_str`, then one can write `Foo.x"literal"` or `Bar.x"literal"` to disambiguate between the two.

Another way to define a macro would be like this:

```
macro foo_str(str, flag)
    # do stuff
end
```

This macro can then be called with the following syntax:

```
foo"str"flag
```

The type of flag in the above mentioned syntax would be a `String` with contents of whatever trails after the string literal.

19.6 Generated functions

A very special macro is `@generated`, which allows you to define so-called *generated functions*. These have the capability to generate specialized code depending on the types of their arguments with more flexibility and/or less code than what can be achieved with multiple dispatch. While macros work with expressions at parse time and cannot access the types of their inputs, a generated function gets expanded at a time when the types of the arguments are known, but the function is not yet compiled.

Instead of performing some calculation or action, a generated function declaration returns a quoted expression which then forms the body for the method corresponding to the types of the arguments. When a generated function is called, the expression it returns is compiled and then run. To make this efficient, the result is usually cached. And to make this inferable, only a limited subset of the language is usable. Thus, generated functions provide a flexible way to move work from run time to compile time, at the expense of greater restrictions on allowed constructs.

When defining generated functions, there are five main differences to ordinary functions:

1. You annotate the function declaration with the `@generated` macro. This adds some information to the AST that lets the compiler know that this is a generated function.
2. In the body of the generated function you only have access to the *types* of the arguments – not their values.
3. Instead of calculating something or performing some action, you return a *quoted expression* which, when evaluated, does what you want.

4. Generated functions are only permitted to call functions that were defined *before* the definition of the generated function. (Failure to follow this may result in getting `MethodErrors` referring to functions from a future world-age.)

5. Generated functions must not *mutate* or *observe* any non-constant global state (including, for example, IO, locks, non-local dictionaries, or using `hasmethod`). This means they can only read global constants, and cannot have any side effects. In other words, they must be completely pure. Due to an implementation limitation, this also means that they currently cannot define a closure or generator.

It's easiest to illustrate this with an example. We can declare a generated function `foo` as

```
julia> @generated function foo(x)
           Core.println(x)
           return :(x * x)
       end
foo (generic function with 1 method)
```

Note that the body returns a quoted expression, namely `:(x * x)`, rather than just the value of `x * x`.

From the caller's perspective, this is identical to a regular function; in fact, you don't have to know whether you're calling a regular or generated function. Let's see how `foo` behaves:

```
julia> x = foo(2); # note: output is from println() statement in the body
Int64

julia> x           # now we print x
4

julia> y = foo("bar");
String

julia> y
"barbar"
```

So, we see that in the body of the generated function, `x` is the *type* of the passed argument, and the value returned by the generated function, is the result of evaluating the quoted expression we returned from the definition, now with the *value* of `x`.

What happens if we evaluate `foo` again with a type that we have already used?

```
julia> foo(4)
16
```

Note that there is no printout of `Int64`. We can see that the body of the generated function was only executed once here, for the specific set of argument types, and the result was cached. After that, for this example, the expression returned from the generated function on the first invocation was re-used as the method body. However, the actual caching behavior is an implementation-defined performance optimization, so it is invalid to depend too closely on this behavior.

The number of times a generated function is generated *might* be only once, but it *might* also be more often, or appear to not happen at all. As a consequence, you should *never* write a generated function with side effects - when, and how often, the side effects occur is undefined. (This is true for macros too - and just like for macros, the use of `eval` in a generated function is a sign that you're doing something the wrong way.) However, unlike macros, the runtime system cannot correctly handle a call to `eval`, so it is disallowed.

It is also important to see how `@generated` functions interact with method redefinition. Following the principle that a correct `@generated` function must not observe any mutable state or cause any mutation of global state, we see the following behavior. Observe that the generated function *cannot* call any method that was not defined prior to the *definition* of the generated function itself.

Initially `f(x)` has one definition

```
julia> f(x) = "original definition";
```

Define other operations that use `f(x)`:

```
julia> g(x) = f(x);

julia> @generated gen1(x) = f(x);

julia> @generated gen2(x) = :(f(x));
```

We now add some new definitions for `f(x)`:

```
julia> f(x::Int) = "definition for Int";

julia> f(x::Type{Int}) = "definition for Type{Int}";
```

and compare how these results differ:

```
julia> f(1)
"definition for Int"

julia> g(1)
"definition for Int"

julia> gen1(1)
"original definition"

julia> gen2(1)
"definition for Int"
```

Each method of a generated function has its own view of defined functions:

```
julia> @generated gen1(x::Real) = f(x);

julia> gen1(1)
"definition for Type{Int}"
```

The example generated function `foo` above did not do anything a normal function `foo(x) = x * x` could not do (except printing the type on the first invocation, and incurring higher overhead). However, the power of a generated function lies in its ability to compute different quoted expressions depending on the types passed to it:

```
julia> @generated function bar(x)
           if x <: Integer
               return :(x ^ 2)
           else
               return :(x)
           end
       end
bar (generic function with 1 method)

julia> bar(4)
16

julia> bar("baz")
"baz"
```

(although of course this contrived example would be more easily implemented using multiple dispatch...)

Abusing this will corrupt the runtime system and cause undefined behavior:

```
julia> @generated function baz(x)
           if rand() < .9
               return :(x^2)
           else
               return :("boo!")
           end
       end
baz (generic function with 1 method)
```

Since the body of the generated function is non-deterministic, its behavior, *and the behavior of all subsequent code* is undefined.

Don't copy these examples!

These examples are hopefully helpful to illustrate how generated functions work, both in the definition end and at the call site; however, *don't copy them*, for the following reasons:

- the `foo` function has side-effects (the call to `Core.println`), and it is undefined exactly when, how often or how many times these side-effects will occur
- the `bar` function solves a problem that is better solved with multiple dispatch - defining `bar(x) = x` and `bar(x::Integer) = x ^ 2` will do the same thing, but it is both simpler and faster.
- the `baz` function is pathological

Note that the set of operations that should not be attempted in a generated function is unbounded, and the runtime system can currently only detect a subset of the invalid operations. There are many other

operations that will simply corrupt the runtime system without notification, usually in subtle ways not obviously connected to the bad definition. Because the function generator is run during inference, it must respect all of the limitations of that code.

Some operations that should not be attempted include:

1. Caching of native pointers.
2. Interacting with the contents or methods of `Core.Compiler` in any way.
3. Observing any mutable state.
    - Inference on the generated function may be run at *any* time, including while your code is attempting to observe or mutate this state.
4. Taking any locks: C code you call out to may use locks internally, (for example, it is not problematic to call `malloc`, even though most implementations require locks internally) but don't attempt to hold or acquire any while executing Julia code.
5. Calling any function that is defined after the body of the generated function. This condition is relaxed for incrementally-loaded precompiled modules to allow calling any function in the module.

Alright, now that we have a better understanding of how generated functions work, let's use them to build some more advanced (and valid) functionality...

An advanced example

Julia's base library has an internal `sub2ind` function to calculate a linear index into an n-dimensional array, based on a set of n multilinear indices - in other words, to calculate the index `i` that can be used to index into an array `A` using `A[i]`, instead of `A[x,y,z,...]`. One possible implementation is the following:

```
julia> function sub2ind_loop(dims::NTuple{N}, I::Integer...) where N
           ind = I[N] - 1
           for i = N-1:-1:1
               ind = I[i]-1 + dims[i]*ind
           end
           return ind + 1
       end;

julia> sub2ind_loop((3, 5), 1, 2)
4
```

The same thing can be done using recursion:

```
julia> sub2ind_rec(dims::Tuple{}) = 1;

julia> sub2ind_rec(dims::Tuple{}, i1::Integer, I::Integer...) =
           i1 == 1 ? sub2ind_rec(dims, I...) : throw(BoundsError());

julia> sub2ind_rec(dims::Tuple{Integer, Vararg{Integer}}, i1::Integer) = i1;
```

```
julia> sub2ind_rec(dims::Tuple{Integer, Vararg{Integer}}, i1::Integer, I::Integer...) =
           i1 + dims[1] * (sub2ind_rec(Base.tail(dims), I...) - 1);

julia> sub2ind_rec((3, 5), 1, 2)
4
```

Both these implementations, although different, do essentially the same thing: a runtime loop over the dimensions of the array, collecting the offset in each dimension into the final index.

However, all the information we need for the loop is embedded in the type information of the arguments. This allows the compiler to move the iteration to compile time and eliminate the runtime loops altogether. We can utilize generated functions to achieve a similar effect; in compiler parlance, we use generated functions to manually unroll the loop. The body becomes almost identical, but instead of calculating the linear index, we build up an *expression* that calculates the index:

```
julia> @generated function sub2ind_gen(dims::NTuple{N}, I::Integer...) where N
           ex = :(I[$N] - 1)
           for i = (N - 1):-1:1
               ex = :(I[$i] - 1 + dims[$i] * $ex)
           end
           return :($ex + 1)
       end;

julia> sub2ind_gen((3, 5), 1, 2)
4
```

What code will this generate?

An easy way to find out is to extract the body into another (regular) function:

```
julia> function sub2ind_gen_impl(dims::Type{T}, I...) where T <: NTuple{N,Any} where N
           length(I) == N || return :(error("partial indexing is unsupported"))
           ex = :(I[$N] - 1)
           for i = (N - 1):-1:1
               ex = :(I[$i] - 1 + dims[$i] * $ex)
           end
           return :($ex + 1)
       end;

julia> @generated function sub2ind_gen(dims::NTuple{N}, I::Integer...) where N
           return sub2ind_gen_impl(dims, I...)
       end;

julia> sub2ind_gen((3, 5), 1, 2)
4
```

We can now execute `sub2ind_gen_impl` and examine the expression it returns:

```
julia> sub2ind_gen_impl(Tuple{Int,Int}, Int, Int)
:(((I[1] - 1) + dims[1] * (I[2] - 1)) + 1)
```

So, the method body that will be used here doesn't include a loop at all - just indexing into the two tuples, multiplication and addition/subtraction. All the looping is performed compile-time, and we avoid looping during execution entirely. Thus, we only loop *once per type*, in this case once per `N` (except in edge cases where the function is generated more than once - see disclaimer above).

Optionally-generated functions

Generated functions can achieve high efficiency at run time, but come with a compile time cost: a new function body must be generated for every combination of concrete argument types. Typically, Julia is able to compile "generic" versions of functions that will work for any arguments, but with generated functions this is impossible. This means that programs making heavy use of generated functions might be impossible to statically compile.

To solve this problem, the language provides syntax for writing normal, non-generated alternative implementations of generated functions. Applied to the `sub2ind` example above, it would look like this:

```
julia> function sub2ind_gen_impl(dims::Type{T}, I...) where T <: NTuple{N,Any} where N
           ex = :(I[$N] - 1)
           for i = (N - 1):-1:1
               ex = :(I[$i] - 1 + dims[$i] * $ex)
           end
           return :($ex + 1)
       end;

julia> function sub2ind_gen_fallback(dims::NTuple{N}, I) where N
           ind = I[N] - 1
           for i = (N - 1):-1:1
               ind = I[i] - 1 + dims[i]*ind
           end
           return ind + 1
       end;

julia> function sub2ind_gen(dims::NTuple{N}, I::Integer...) where N
           length(I) == N || error("partial indexing is unsupported")
           if @generated
               return sub2ind_gen_impl(dims, I...)
           else
               return sub2ind_gen_fallback(dims, I)
           end
       end;

julia> sub2ind_gen((3, 5), 1, 2)
4
```

Internally, this code creates two implementations of the function: a generated one where the first block in `if @generated` is used, and a normal one where the `else` block is used. Inside the `then` part of the `if @generated` block, code has the same semantics as other generated functions: argument names refer to types, and the code should return an expression. Multiple `if @generated` blocks may occur, in which case the generated implementation uses all of the `then` blocks and the alternate implementation uses all of the `else` blocks.

Notice that we added an error check to the top of the function. This code will be common to both versions, and is run-time code in both versions (it will be quoted and returned as an expression from the generated

version). That means that the values and types of local variables are not available at code generation time — the code-generation code can only see the types of arguments.

In this style of definition, the code generation feature is essentially an optional optimization. The compiler will use it if convenient, but otherwise may choose to use the normal implementation instead. This style is preferred, since it allows the compiler to make more decisions and compile programs in more ways, and since normal code is more readable than code-generating code. However, which implementation is used depends on compiler implementation details, so it is essential for the two implementations to behave identically.

Chapter 20

Single- and multi-dimensional Arrays

Julia, like most technical computing languages, provides a first-class array implementation. Most technical computing languages pay a lot of attention to their array implementation at the expense of other containers. Julia does not treat arrays in any special way. The array library is implemented almost completely in Julia itself, and derives its performance from the compiler, just like any other code written in Julia. As such, it's also possible to define custom array types by inheriting from `AbstractArray`. See the manual section on the AbstractArray interface for more details on implementing a custom array type.

An array is a collection of objects stored in a multi-dimensional grid. Zero-dimensional arrays are allowed, see this FAQ entry. In the most general case, an array may contain objects of type `Any`. For most computational purposes, arrays should contain objects of a more specific type, such as `Float64` or `Int32`.

In general, unlike many other technical computing languages, Julia does not expect programs to be written in a vectorized style for performance. Julia's compiler uses type inference and generates optimized code for scalar array indexing, allowing programs to be written in a style that is convenient and readable, without sacrificing performance, and using less memory at times.

In Julia, all arguments to functions are passed by sharing (i.e. by pointers). Some technical computing languages pass arrays by value, and while this prevents accidental modification by callees of a value in the caller, it makes avoiding unwanted copying of arrays difficult. By convention, a function name ending with a `!` indicates that it will mutate or destroy the value of one or more of its arguments (compare, for example, `sort` and `sort!`). Callees must make explicit copies to ensure that they don't modify inputs that they don't intend to change. Many non-mutating functions are implemented by calling a function of the same name with an added `!` at the end on an explicit copy of the input, and returning that copy.

20.1 Basic Functions

20.2 Construction and Initialization

Many functions for constructing and initializing arrays are provided. In the following list of such functions, calls with a `dims...` argument can either take a single tuple of dimension sizes or a series of dimension sizes passed as a variable number of arguments. Most of these functions also accept a first input T, which is the element type of the array. If the type T is omitted it will default to `Float64`.

To see the various ways we can pass dimensions to these functions, consider the following examples:

[1] *iid*, independently and identically distributed.

Function	Description
`eltype(A)`	the type of the elements contained in `A`
`length(A)`	the number of elements in `A`
`ndims(A)`	the number of dimensions of `A`
`size(A)`	a tuple containing the dimensions of `A`
`size(A,n)`	the size of `A` along dimension `n`
`axes(A)`	a tuple containing the valid indices of `A`
`axes(A,n)`	a range expressing the valid indices along dimension `n`
`eachindex(A)`	an efficient iterator for visiting each position in `A`
`stride(A,k)`	the stride (linear index distance between adjacent elements) along dimension `k`
`strides(A)`	a tuple of the strides in each dimension

```
julia> zeros(Int8, 2, 3)
2×3 Matrix{Int8}:
 0  0  0
 0  0  0

julia> zeros(Int8, (2, 3))
2×3 Matrix{Int8}:
 0  0  0
 0  0  0

julia> zeros((2, 3))
2×3 Matrix{Float64}:
 0.0  0.0  0.0
 0.0  0.0  0.0
```

Here, `(2, 3)` is a `Tuple` and the first argument — the element type — is optional, defaulting to `Float64`.

20.3 Array literals

Arrays can also be directly constructed with square braces; the syntax `[A, B, C, ...]` creates a one-dimensional array (i.e., a vector) containing the comma-separated arguments as its elements. The element type (`eltype`) of the resulting array is automatically determined by the types of the arguments inside the braces. If all the arguments are the same type, then that is its `eltype`. If they all have a common promotion type then they get converted to that type using `convert` and that type is the array's `eltype`. Otherwise, a heterogeneous array that can hold anything — a `Vector{Any}` — is constructed; this includes the literal `[]` where no arguments are given. Array literals can be typed with the syntax `T[A, B, C, ...]` where `T` is a type.

```
julia> [1, 2, 3] # An array of `Int`s
3-element Vector{Int64}:
 1
 2
 3

julia> promote(1, 2.3, 4//5) # This combination of Int, Float64 and Rational promotes to Float64
(1.0, 2.3, 0.8)
```

Function	Description
`Array{T}(undef, dims...)`	an uninitialized dense `Array`
`zeros(T, dims...)`	an `Array` of all zeros
`ones(T, dims...)`	an `Array` of all ones
`trues(dims...)`	a `BitArray` with all values `true`
`falses(dims...)`	a `BitArray` with all values `false`
`reshape(A, dims...)`	an array containing the same data as `A`, but with different dimensions
`copy(A)`	copy `A`
`deepcopy(A)`	copy `A`, recursively copying its elements
`similar(A, T, dims...)`	an uninitialized array of the same type as `A` (dense, sparse, etc.), but with the specified element type and dimensions. The second and third arguments are both optional, defaulting to the element type and dimensions of `A` if omitted.
`reinterpret(T, A)`	an array with the same binary data as `A`, but with element type `T`
`rand(T, dims...)`	an `Array` with random, iid [1] and uniformly distributed values. For floating point types `T`, the values lie in the half-open interval $[0, 1)$.
`randn(T, dims...)`	an `Array` with random, iid and standard normally distributed values
`Matrix{T}(I, m, n)`	`m`-by-`n` identity matrix. Requires `using LinearAlgebra` for `I`.
`range(start, stop, n)`	a range of `n` linearly spaced elements from `start` to `stop`
`fill!(A, x)`	fill the array `A` with the value `x`
`fill(x, dims...)`	an `Array` filled with the value `x`. In particular, `fill(x)` constructs a zero-dimensional `Array` containing `x`.

```
julia> [1, 2.3, 4//5] # Thus that's the element type of this Array
3-element Vector{Float64}:
 1.0
 2.3
 0.8

julia> Float32[1, 2.3, 4//5] # Specify element type manually
3-element Vector{Float32}:
 1.0
 2.3
 0.8

julia> []
Any[]
```

Concatenation

If the arguments inside the square brackets are separated by single semicolons (`;`) or newlines instead of commas, then their contents are *vertically concatenated* together instead of the arguments being used as

elements themselves.

```
julia> [1:2, 4:5] # Has a comma, so no concatenation occurs. The ranges are themselves the
↪  elements
2-element Vector{UnitRange{Int64}}:
 1:2
 4:5

julia> [1:2; 4:5]
4-element Vector{Int64}:
 1
 2
 4
 5

julia> [1:2
        4:5
        6]
5-element Vector{Int64}:
 1
 2
 4
 5
 6
```

Similarly, if the arguments are separated by tabs or spaces or double semicolons, then their contents are *horizontally concatenated* together.

```
julia> [1:2  4:5  7:8]
2×3 Matrix{Int64}:
 1  4  7
 2  5  8

julia> [[1,2]  [4,5]  [7,8]]
2×3 Matrix{Int64}:
 1  4  7
 2  5  8

julia> [1 2 3] # Numbers can also be horizontally concatenated
1×3 Matrix{Int64}:
 1  2  3

julia> [1;; 2;; 3;; 4]
1×4 Matrix{Int64}:
 1  2  3  4
```

Single semicolons (or newlines) and spaces (or tabs) can be combined to concatenate both horizontally and vertically at the same time.

```
julia> [1 2
        3 4]
```

```
2×2 Matrix{Int64}:
 1  2
 3  4

julia> [zeros(Int, 2, 2) [1; 2]
        [3 4]            5]
3×3 Matrix{Int64}:
 0  0  1
 0  0  2
 3  4  5

julia> [[1 1]; 2 3; [4 4]]
3×2 Matrix{Int64}:
 1  1
 2  3
 4  4
```

Spaces (and tabs) have a higher precedence than semicolons, performing any horizontal concatenations first and then concatenating the result. Using double semicolons for the horizontal concatenation, on the other hand, performs any vertical concatenations before horizontally concatenating the result.

```
julia> [zeros(Int, 2, 2) ; [3 4] ;; [1; 2] ; 5]
3×3 Matrix{Int64}:
 0  0  1
 0  0  2
 3  4  5

julia> [1:2; 4;; 1; 3:4]
3×2 Matrix{Int64}:
 1  1
 2  3
 4  4
```

Just as `;` and `;;` concatenate in the first and second dimension, using more semicolons extends this same general scheme. The number of semicolons in the separator specifies the particular dimension, so `;;;` concatenates in the third dimension, `;;;;` in the 4th, and so on. Fewer semicolons take precedence, so the lower dimensions are generally concatenated first.

```
julia> [1; 2;; 3; 4;; 5; 6;;;
        7; 8;; 9; 10;; 11; 12]
2×3×2 Array{Int64, 3}:
[:, :, 1] =
 1  3  5
 2  4  6

[:, :, 2] =
 7   9  11
 8  10  12
```

Like before, spaces (and tabs) for horizontal concatenation have a higher precedence than any number of semicolons. Thus, higher-dimensional arrays can also be written by specifying their rows first, with their elements textually arranged in a manner similar to their layout:

```
julia> [1 3 5
        2 4 6;;;
        7 9 11
        8 10 12]
2×3×2 Array{Int64, 3}:
[:, :, 1] =
 1  3  5
 2  4  6

[:, :, 2] =
 7   9  11
 8  10  12

julia> [1 2;;; 3 4;;;; 5 6;;; 7 8]
1×2×2×2 Array{Int64, 4}:
[:, :, 1, 1] =
 1  2

[:, :, 2, 1] =
 3  4

[:, :, 1, 2] =
 5  6

[:, :, 2, 2] =
 7  8

julia> [[1 2;;; 3 4];;;; [5 6];;; [7 8]]
1×2×2×2 Array{Int64, 4}:
[:, :, 1, 1] =
 1  2

[:, :, 2, 1] =
 3  4

[:, :, 1, 2] =
 5  6

[:, :, 2, 2] =
 7  8
```

Although they both mean concatenation in the second dimension, spaces (or tabs) and `;;` cannot appear in the same array expression unless the double semicolon is simply serving as a "line continuation" character. This allows a single horizontal concatenation to span multiple lines (without the line break being interpreted as a vertical concatenation).

```
julia> [1 2 ;;
       3 4]
1×4 Matrix{Int64}:
 1  2  3  4
```

Terminating semicolons may also be used to add trailing length 1 dimensions.

```
julia> [1;;]
1×1 Matrix{Int64}:
 1

julia> [2; 3;;;]
2×1×1 Array{Int64, 3}:
[:, :, 1] =
 2
 3
```

More generally, concatenation can be accomplished through the `cat` function. These syntaxes are shorthands for function calls that themselves are convenience functions:

Syntax	Function	Description
	`cat`	concatenate input arrays along dimension(s) k
`[A; B; C; ...]`	`vcat`	shorthand for `cat(A...; dims=1)`
`[A B C ...]`	`hcat`	shorthand for `cat(A...; dims=2)`
`[A B; C D; ...]`	`hvcat`	simultaneous vertical and horizontal concatenation
`[A; C;; B; D;;; ...]`	`hvncat`	simultaneous n-dimensional concatenation, where number of semicolons indicate the dimension to concatenate

Typed array literals

An array with a specific element type can be constructed using the syntax `T[A, B, C, ...]`. This will construct a 1-d array with element type `T`, initialized to contain elements `A`, `B`, `C`, etc. For example, `Any[x, y, z]` constructs a heterogeneous array that can contain any values.

Concatenation syntax can similarly be prefixed with a type to specify the element type of the result.

```
julia> [[1 2] [3 4]]
1×4 Matrix{Int64}:
 1  2  3  4

julia> Int8[[1 2] [3 4]]
1×4 Matrix{Int8}:
 1  2  3  4
```

20.4 Comprehensions

Comprehensions provide a general and powerful way to construct arrays. Comprehension syntax is similar to set construction notation in mathematics:

```
A = [ F(x, y, ...) for x=rx, y=ry, ... ]
```

The meaning of this form is that `F(x,y,...)` is evaluated with the variables `x`, `y`, etc. taking on each value in their given list of values. Values can be specified as any iterable object, but will commonly be ranges

like `1:n` or `2:(n-1)`, or explicit arrays of values like `[1.2, 3.4, 5.7]`. The result is an N-d dense array with dimensions that are the concatenation of the dimensions of the variable ranges `rx`, `ry`, etc. and each `F(x,y,...)` evaluation returns a scalar.

The following example computes a weighted average of the current element and its left and right neighbor along a 1-d grid:

```
julia> x = [4, 8, 2, 6, 10, 10, 2, 8]
8-element Vector{Int64}:
  4
  8
  2
  6
 10
 10
  2
  8

julia> [ 0.25*x[i-1] + 0.5*x[i] + 0.25*x[i+1] for i=2:length(x)-1 ]
6-element Vector{Float64}:
 5.5
 4.5
 6.0
 9.0
 8.0
 5.5
```

The resulting array type depends on the types of the computed elements just like array literals do. In order to control the type explicitly, a type can be prepended to the comprehension. For example, we could have requested the result in single precision by writing:

```
Float32[ 0.25*x[i-1] + 0.5*x[i] + 0.25*x[i+1] for i=2:length(x)-1 ]
```

20.5 Generator Expressions

Comprehensions can also be written without the enclosing square brackets, producing an object known as a generator. This object can be iterated to produce values on demand, instead of allocating an array and storing them in advance (see Iteration). For example, the following expression sums a series without allocating memory:

```
julia> sum(1/n^2 for n=1:1000)
1.6439345666815615
```

When writing a generator expression with multiple dimensions inside an argument list, parentheses are needed to separate the generator from subsequent arguments:

```
julia> map(tuple, 1/(i+j) for i=1:2, j=1:2, [1:4;])
ERROR: ParseError:
# Error @ none:1:44
map(tuple, 1/(i+j) for i=1:2, j=1:2, [1:4;])
#                                            └ ── invalid iteration spec: expected one of `=` `in`
↪  or `∈`
```

All comma-separated expressions after `for` are interpreted as ranges. Adding parentheses lets us add a third argument to `map`:

```
julia> map(tuple, (1/(i+j) for i=1:2, j=1:2), [1 3; 2 4])
2×2 Matrix{Tuple{Float64, Int64}}:
 (0.5, 1)       (0.333333, 3)
 (0.333333, 2)  (0.25, 4)
```

Generators are implemented via inner functions. Just like inner functions used elsewhere in the language, variables from the enclosing scope can be "captured" in the inner function. For example, `sum(p[i] - q[i] for i=1:n)` captures the three variables `p`, `q` and `n` from the enclosing scope. Captured variables can present performance challenges; see performance tips.

Ranges in generators and comprehensions can depend on previous ranges by writing multiple `for` keywords:

```
julia> [(i, j) for i=1:3 for j=1:i]
6-element Vector{Tuple{Int64, Int64}}:
 (1, 1)
 (2, 1)
 (2, 2)
 (3, 1)
 (3, 2)
 (3, 3)
```

In such cases, the result is always 1-d.

Generated values can be filtered using the `if` keyword:

```
julia> [(i, j) for i=1:3 for j=1:i if i+j == 4]
2-element Vector{Tuple{Int64, Int64}}:
 (2, 2)
 (3, 1)
```

20.6 Indexing

The general syntax for indexing into an n-dimensional array `A` is:

```
X = A[I_1, I_2, ..., I_n]
```

where each `I_k` may be a scalar integer, an array of integers, or any other supported index. This includes `Colon` (`:`) to select all indices within the entire dimension, ranges of the form `a:c` or `a:b:c` to select contiguous or strided subsections, and arrays of booleans to select elements at their `true` indices.

If all the indices are scalars, then the result `X` is a single element from the array `A`. Otherwise, `X` is an array with the same number of dimensions as the sum of the dimensionalities of all the indices.

If all indices `I_k` are vectors, for example, then the shape of `X` would be `(length(I_1), length(I_2), ..., length(I_n))`, with location `i_1, i_2, ..., i_n` of `X` containing the value `A[I_1[i_1], I_2[i_2], ..., I_n[i_n]]`.

Example:

```
julia> A = reshape(collect(1:16), (2, 2, 2, 2))
2×2×2×2 Array{Int64, 4}:
[:, :, 1, 1] =
 1  3
 2  4

[:, :, 2, 1] =
 5  7
 6  8

[:, :, 1, 2] =
  9  11
 10  12

[:, :, 2, 2] =
 13  15
 14  16

julia> A[1, 2, 1, 1] # all scalar indices
3

julia> A[[1, 2], [1], [1, 2], [1]] # all vector indices
2×1×2×1 Array{Int64, 4}:
[:, :, 1, 1] =
 1
 2

[:, :, 2, 1] =
 5
 6

julia> A[[1, 2], [1], [1, 2], 1] # a mix of index types
2×1×2 Array{Int64, 3}:
[:, :, 1] =
 1
 2

[:, :, 2] =
 5
 6
```

Note how the size of the resulting array is different in the last two cases.

If `I_1` is changed to a two-dimensional matrix, then `X` becomes an n+1-dimensional array of shape `(size(I_1, 1), size(I_1, 2), length(I_2), ..., length(I_n))`. The matrix adds a dimension.

Example:

```
julia> A = reshape(collect(1:16), (2, 2, 2, 2));

julia> A[[1 2; 1 2]]
2×2 Matrix{Int64}:
 1  2
```

```
 1  2

julia> A[[1 2; 1 2], 1, 2, 1]
2×2 Matrix{Int64}:
 5  6
 5  6
```

The location `i_1, i_2, i_3, ..., i_{n+1}` contains the value at `A[I_1[i_1, i_2], I_2[i_3], ..., I_n[i_{n+1}]]`. All dimensions indexed with scalars are dropped. For example, if `J` is an array of indices, then the result of `A[2, J, 3]` is an array with size `size(J)`. Its `j`th element is populated by `A[2, J[j], 3]`.

As a special part of this syntax, the end keyword may be used to represent the last index of each dimension within the indexing brackets, as determined by the size of the innermost array being indexed. Indexing syntax without the end keyword is equivalent to a call to `getindex`:

```
X = getindex(A, I_1, I_2, ..., I_n)
```

Example:

```
julia> x = reshape(1:16, 4, 4)
4×4 reshape(::UnitRange{Int64}, 4, 4) with eltype Int64:
 1  5   9  13
 2  6  10  14
 3  7  11  15
 4  8  12  16

julia> x[2:3, 2:end-1]
2×2 Matrix{Int64}:
 6  10
 7  11

julia> x[1, [2 3; 4 1]]
2×2 Matrix{Int64}:
  5  9
 13  1
```

20.7 Indexed Assignment

The general syntax for assigning values in an n-dimensional array `A` is:

```
A[I_1, I_2, ..., I_n] = X
```

where each `I_k` may be a scalar integer, an array of integers, or any other supported index. This includes `Colon` (`:`) to select all indices within the entire dimension, ranges of the form `a:c` or `a:b:c` to select contiguous or strided subsections, and arrays of booleans to select elements at their `true` indices.

If all indices `I_k` are integers, then the value in location `I_1, I_2, ..., I_n` of `A` is overwritten with the value of `X`, converting to the `eltype` of `A` if necessary.

If any index `I_k` is itself an array, then the right hand side `X` must also be an array with the same shape as the result of indexing `A[I_1, I_2, ..., I_n]` or a vector with the same number of elements. The value in location `I_1[i_1], I_2[i_2], ..., I_n[i_n]` of `A` is overwritten with the value `X[i_1, i_2, ..., i_n]`, converting if necessary. The element-wise assignment operator `.=` may be used to broadcast `X` across the selected locations:

```
A[I_1, I_2, ..., I_n] .= X
```

Just as in Indexing, the end keyword may be used to represent the last index of each dimension within the indexing brackets, as determined by the size of the array being assigned into. Indexed assignment syntax without the end keyword is equivalent to a call to `setindex!`:

```
setindex!(A, X, I_1, I_2, ..., I_n)
```

Example:

```
julia> x = collect(reshape(1:9, 3, 3))
3×3 Matrix{Int64}:
 1  4  7
 2  5  8
 3  6  9

julia> x[3, 3] = -9;

julia> x[1:2, 1:2] = [-1 -4; -2 -5];

julia> x
3×3 Matrix{Int64}:
 -1  -4   7
 -2  -5   8
  3   6  -9
```

20.8 Supported index types

In the expression `A[I_1, I_2, ..., I_n]`, each `I_k` may be a scalar index, an array of scalar indices, or an object that represents an array of scalar indices and can be converted to such by `to_indices`:

1. A scalar index. By default this includes:
   - Non-boolean integers
   - `CartesianIndex{N}`s, which behave like an `N`-tuple of integers spanning multiple dimensions (see below for more details)
2. An array of scalar indices. This includes:
   - Vectors and multidimensional arrays of integers
   - Empty arrays like `[]`, which select no elements e.g. `A[[]]` (not to be confused with `A[]`)
   - Ranges like `a:c` or `a:b:c`, which select contiguous or strided subsections from `a` to `c` (inclusive)

   - Any custom array of scalar indices that is a subtype of `AbstractArray`
   - Arrays of `CartesianIndex{N}` (see below for more details)

3. An object that represents an array of scalar indices and can be converted to such by `to_indices`. By default this includes:

   - `Colon()` (`:`), which represents all indices within an entire dimension or across the entire array
   - Arrays of booleans, which select elements at their `true` indices (see below for more details)

Some examples:

```
julia> A = reshape(collect(1:2:18), (3, 3))
3×3 Matrix{Int64}:
 1   7  13
 3   9  15
 5  11  17

julia> A[4]
7

julia> A[[2, 5, 8]]
3-element Vector{Int64}:
  3
  9
 15

julia> A[[1 4; 3 8]]
2×2 Matrix{Int64}:
 1   7
 5  15

julia> A[[]]
Int64[]

julia> A[1:2:5]
3-element Vector{Int64}:
 1
 5
 9

julia> A[2, :]
3-element Vector{Int64}:
  3
  9
 15

julia> A[:, 3]
3-element Vector{Int64}:
 13
 15
 17
```

```
julia> A[:, 3:3]
3×1 Matrix{Int64}:
 13
 15
 17
```

Cartesian indices

The special `CartesianIndex{N}` object represents a scalar index that behaves like an `N`-tuple of integers spanning multiple dimensions. For example:

```
julia> A = reshape(1:32, 4, 4, 2);

julia> A[3, 2, 1]
7

julia> A[CartesianIndex(3, 2, 1)] == A[3, 2, 1] == 7
true
```

Considered alone, this may seem relatively trivial; `CartesianIndex` simply gathers multiple integers together into one object that represents a single multidimensional index. When combined with other indexing forms and iterators that yield `CartesianIndex`es, however, this can produce very elegant and efficient code. See Iteration below, and for some more advanced examples, see this blog post on multidimensional algorithms and iteration.

Arrays of `CartesianIndex{N}` are also supported. They represent a collection of scalar indices that each span `N` dimensions, enabling a form of indexing that is sometimes referred to as pointwise indexing. For example, it enables accessing the diagonal elements from the first "page" of `A` from above:

```
julia> page = A[:, :, 1]
4×4 Matrix{Int64}:
 1  5   9  13
 2  6  10  14
 3  7  11  15
 4  8  12  16

julia> page[[CartesianIndex(1, 1),
             CartesianIndex(2, 2),
             CartesianIndex(3, 3),
             CartesianIndex(4, 4)]]
4-element Vector{Int64}:
  1
  6
 11
 16
```

This can be expressed much more simply with dot broadcasting and by combining it with a normal integer index (instead of extracting the first page from `A` as a separate step). It can even be combined with a `:` to extract both diagonals from the two pages at the same time:

```
julia> A[CartesianIndex.(axes(A, 1), axes(A, 2)), 1]
4-element Vector{Int64}:
  1
  6
 11
 16

julia> A[CartesianIndex.(axes(A, 1), axes(A, 2)), :]
4×2 Matrix{Int64}:
  1  17
  6  22
 11  27
 16  32
```

Warning

`CartesianIndex` and arrays of `CartesianIndex` are not compatible with the end keyword to represent the last index of a dimension. Do not use end in indexing expressions that may contain either `CartesianIndex` or arrays thereof.

Logical indexing

Often referred to as logical indexing or indexing with a logical mask, indexing by a boolean array selects elements at the indices where its values are `true`. Indexing by a boolean vector `B` is effectively the same as indexing by the vector of integers that is returned by `findall(B)`. Similarly, indexing by a `N`-dimensional boolean array is effectively the same as indexing by the vector of `CartesianIndex{N}`s where its values are `true`. A logical index must be an array of the same shape as the dimension(s) it indexes into, or it must be the only index provided and match the shape of the one-dimensional reshaped view of the array it indexes into. It is generally more efficient to use boolean arrays as indices directly instead of first calling `findall`.

```
julia> x = reshape(1:12, 2, 3, 2)
2×3×2 reshape(::UnitRange{Int64}, 2, 3, 2) with eltype Int64:
[:, :, 1] =
 1  3  5
 2  4  6

[:, :, 2] =
 7   9  11
 8  10  12

julia> x[:, [true false; false true; true false]]
2×3 Matrix{Int64}:
 1  5   9
 2  6  10

julia> mask = map(ispow2, x)
2×3×2 Array{Bool, 3}:
[:, :, 1] =
 1  0  0
 1  1  0
```

```
[:, :, 2] =
 0  0  0
 1  0  0

julia> x[mask]
4-element Vector{Int64}:
 1
 2
 4
 8

julia> x[vec(mask)] == x[mask] # we can also index with a single Boolean vector
true
```

Number of indices

Cartesian indexing

The ordinary way to index into an `N`-dimensional array is to use exactly `N` indices; each index selects the position(s) in its particular dimension. For example, in the three-dimensional array `A = rand(4, 3, 2)`, `A[2, 3, 1]` will select the number in the second row of the third column in the first "page" of the array. This is often referred to as *cartesian indexing*.

Linear indexing

When exactly one index `i` is provided, that index no longer represents a location in a particular dimension of the array. Instead, it selects the `i`th element using the column-major iteration order that linearly spans the entire array. This is known as *linear indexing*. It essentially treats the array as though it had been reshaped into a one-dimensional vector with `vec`.

```
julia> A = [2 6; 4 7; 3 1]
3×2 Matrix{Int64}:
 2  6
 4  7
 3  1

julia> A[5]
7

julia> vec(A)[5]
7
```

A linear index into the array `A` can be converted to a `CartesianIndex` for cartesian indexing with `CartesianIndices(A)[i]` (see `CartesianIndices`), and a set of `N` cartesian indices can be converted to a linear index with `LinearIndices(A)[i_1, i_2, ..., i_N]` (see `LinearIndices`).

```
julia> CartesianIndices(A)[5]
CartesianIndex(2, 2)

julia> LinearIndices(A)[2, 2]
5
```

It's important to note that there's a very large asymmetry in the performance of these conversions. Converting a linear index to a set of cartesian indices requires dividing and taking the remainder, whereas going the other way is just multiplies and adds. In modern processors, integer division can be 10-50 times slower than multiplication. While some arrays — like `Array` itself — are implemented using a linear chunk of memory and directly use a linear index in their implementations, other arrays — like `Diagonal` — need the full set of cartesian indices to do their lookup (see `IndexStyle` to introspect which is which).

Warning

When iterating over all the indices for an array, it is better to iterate over `eachindex(A)` instead of `1:length(A)`. Not only will this be faster in cases where `A` is `IndexCartesian`, but it will also support arrays with custom indexing, such as OffsetArrays. If only the values are needed, then is better to just iterate the array directly, i.e. `for a in A`.

Omitted and extra indices

In addition to linear indexing, an `N`-dimensional array may be indexed with fewer or more than `N` indices in certain situations.

Indices may be omitted if the trailing dimensions that are not indexed into are all length one. In other words, trailing indices can be omitted only if there is only one possible value that those omitted indices could be for an in-bounds indexing expression. For example, a four-dimensional array with size `(3, 4, 2, 1)` may be indexed with only three indices as the dimension that gets skipped (the fourth dimension) has length one. Note that linear indexing takes precedence over this rule.

```
julia> A = reshape(1:24, 3, 4, 2, 1)
3×4×2×1 reshape(::UnitRange{Int64}, 3, 4, 2, 1) with eltype Int64:
[:, :, 1, 1] =
 1  4  7  10
 2  5  8  11
 3  6  9  12

[:, :, 2, 1] =
 13  16  19  22
 14  17  20  23
 15  18  21  24

julia> A[1, 3, 2] # Omits the fourth dimension (length 1)
19

julia> A[1, 3] # Attempts to omit dimensions 3 & 4 (lengths 2 and 1)
ERROR: BoundsError: attempt to access 3×4×2×1 reshape(::UnitRange{Int64}, 3, 4, 2, 1) with eltype
↪  Int64 at index [1, 3]

julia> A[19] # Linear indexing
19
```

When omitting *all* indices with `A[]`, this semantic provides a simple idiom to retrieve the only element in an array and simultaneously ensure that there was only one element.

Similarly, more than `N` indices may be provided if all the indices beyond the dimensionality of the array are 1 (or more generally are the first and only element of `axes(A, d)` where `d` is that particular dimension number). This allows vectors to be indexed like one-column matrices, for example:

```
julia> A = [8, 6, 7]
3-element Vector{Int64}:
 8
 6
 7

julia> A[2, 1]
6
```

20.9 Iteration

The recommended ways to iterate over a whole array are

```
for a in A
    # Do something with the element a
end

for i in eachindex(A)
    # Do something with i and/or A[i]
end
```

The first construct is used when you need the value, but not index, of each element. In the second construct, `i` will be an `Int` if `A` is an array type with fast linear indexing; otherwise, it will be a `CartesianIndex`:

```
julia> A = rand(4, 3);

julia> B = view(A, 1:3, 2:3);

julia> for i in eachindex(B)
           @show i
       end
i = CartesianIndex(1, 1)
i = CartesianIndex(2, 1)
i = CartesianIndex(3, 1)
i = CartesianIndex(1, 2)
i = CartesianIndex(2, 2)
i = CartesianIndex(3, 2)
```

Note

In contrast with `for i = 1:length(A)`, iterating with `eachindex` provides an efficient way to iterate over any array type. Besides, this also supports generic arrays with custom indexing such as OffsetArrays.

20.10 Array traits

If you write a custom `AbstractArray` type, you can specify that it has fast linear indexing using

```
Base.IndexStyle(::Type{<:MyArray}) = IndexLinear()
```

This setting will cause `eachindex` iteration over a `MyArray` to use integers. If you don't specify this trait, the default value `IndexCartesian()` is used.

20.11 Array and Vectorized Operators and Functions

The following operators are supported for arrays:

1. Unary arithmetic – `-`, `+`
2. Binary arithmetic – `-`, `+`, `*`, `/`, `\`, `^`
3. Comparison – `==`, `!=`, ≈ (`isapprox`), ≉

To enable convenient vectorization of mathematical and other operations, Julia provides the dot syntax `f.(args...)`, e.g. `sin.(x)` or `min.(x, y)`, for elementwise operations over arrays or mixtures of arrays and scalars (a Broadcasting operation); these have the additional advantage of "fusing" into a single loop when combined with other dot calls, e.g. `sin.(cos.(x))`.

Also, *every* binary operator supports a dot version that can be applied to arrays (and combinations of arrays and scalars) in such fused broadcasting operations, e.g. `z .== sin.(x .* y)`.

Note that comparisons such as `==` operate on whole arrays, giving a single boolean answer. Use dot operators like `.==` for elementwise comparisons. (For comparison operations like `<`, *only* the elementwise `.<` version is applicable to arrays.)

Also notice the difference between `max.(a,b)`, which `broadcast`s `max` elementwise over `a` and `b`, and `maximum(a)`, which finds the largest value within `a`. The same relationship holds for `min.(a, b)` and `minimum(a)`.

20.12 Broadcasting

It is sometimes useful to perform element-by-element binary operations on arrays of different sizes, such as adding a vector to each column of a matrix. An inefficient way to do this would be to replicate the vector to the size of the matrix:

```
julia> a = [0.2, 0.5]; A = [1.0 1.6 1.05; 1.07 1.36 1.18];

julia> repeat(a, 1, 3) + A
2×3 Matrix{Float64}:
 1.2   1.8   1.25
 1.57  1.86  1.68
```

This is wasteful when dimensions get large, so Julia provides `broadcast`, which expands singleton dimensions in array arguments to match the corresponding dimension in the other array without using extra memory, and applies the given function elementwise:

```
julia> broadcast(+, a, A)
2×3 Matrix{Float64}:
 1.2   1.8   1.25
 1.57  1.86  1.68

julia> b = [0.9 0.1]
1×2 Matrix{Float64}:
 0.9  0.1

julia> broadcast(+, a, b)
2×2 Matrix{Float64}:
 1.1  0.3
 1.4  0.6
```

Dotted operators such as `.+` and `.*` are equivalent to `broadcast` calls (except that they fuse, as described above). There is also a `broadcast!` function to specify an explicit destination (which can also be accessed in a fusing fashion by `.=` assignment). In fact, `f.(args...)` is equivalent to `broadcast(f, args...)`, providing a convenient syntax to broadcast any function (dot syntax). Nested "dot calls" `f.(...)` (including calls to `.+` etcetera) automatically fuse into a single `broadcast` call.

Additionally, `broadcast` is not limited to arrays (see the function documentation); it also handles scalars, tuples and other collections. By default, only some argument types are considered scalars, including (but not limited to) `Numbers`, `Strings`, `Symbols`, `Types`, `Functions` and some common singletons like `missing` and `nothing`. All other arguments are iterated over or indexed into elementwise.

```
julia> convert.(Float32, [1, 2])
2-element Vector{Float32}:
 1.0
 2.0

julia> ceil.(UInt8, [1.2 3.4; 5.6 6.7])
2×2 Matrix{UInt8}:
 0x02  0x04
 0x06  0x07

julia> string.(1:3, ". ", ["First", "Second", "Third"])
3-element Vector{String}:
 "1. First"
 "2. Second"
 "3. Third"
```

Sometimes, you want a container (like an array) that would normally participate in broadcast to be "protected" from broadcast's behavior of iterating over all of its elements. By placing it inside another container (like a single element `Tuple`) broadcast will treat it as a single value.

```
julia> ([1, 2, 3], [4, 5, 6]) .+ ([1, 2, 3],)
([2, 4, 6], [5, 7, 9])

julia> ([1, 2, 3], [4, 5, 6]) .+ tuple([1, 2, 3])
([2, 4, 6], [5, 7, 9])
```

20.13 Implementation

The base array type in Julia is the abstract type `AbstractArray{T,N}`. It is parameterized by the number of dimensions `N` and the element type `T`. `AbstractVector` and `AbstractMatrix` are aliases for the 1-d and 2-d cases. Operations on `AbstractArray` objects are defined using higher level operators and functions, in a way that is independent of the underlying storage. These operations generally work correctly as a fallback for any specific array implementation.

The `AbstractArray` type includes anything vaguely array-like, and implementations of it might be quite different from conventional arrays. For example, elements might be computed on request rather than stored. However, any concrete `AbstractArray{T,N}` type should generally implement at least `size(A)` (returning an `Int` tuple), `getindex(A, i)` and `getindex(A, i1, ..., iN)`; mutable arrays should also implement `setindex!`. It is recommended that these operations have nearly constant time complexity, as otherwise some array functions may be unexpectedly slow. Concrete types should also typically provide a `similar(A, T=eltype(A), dims=size(A))` method, which is used to allocate a similar array for `copy` and other out-of-place operations. No matter how an `AbstractArray{T,N}` is represented internally, `T` is the type of object returned by *integer* indexing (`A[1, ..., 1]`, when `A` is not empty) and `N` should be the length of the tuple returned by `size`. For more details on defining custom `AbstractArray` implementations, see the array interface guide in the interfaces chapter.

`DenseArray` is an abstract subtype of `AbstractArray` intended to include all arrays where elements are stored contiguously in column-major order (see additional notes in Performance Tips). The `Array` type is a specific instance of `DenseArray`; `Vector` and `Matrix` are aliases for the 1-d and 2-d cases. Very few operations are implemented specifically for `Array` beyond those that are required for all `AbstractArrays`; much of the array library is implemented in a generic manner that allows all custom arrays to behave similarly.

`SubArray` is a specialization of `AbstractArray` that performs indexing by sharing memory with the original array rather than by copying it. A `SubArray` is created with the `view` function, which is called the same way as `getindex` (with an array and a series of index arguments). The result of `view` looks the same as the result of `getindex`, except the data is left in place. `view` stores the input index vectors in a `SubArray` object, which can later be used to index the original array indirectly. By putting the `@views` macro in front of an expression or block of code, any `array[...]` slice in that expression will be converted to create a `SubArray` view instead.

`BitArray`s are space-efficient "packed" boolean arrays, which store one bit per boolean value. They can be used similarly to `Array{Bool}` arrays (which store one byte per boolean value), and can be converted to/from the latter via `Array(bitarray)` and `BitArray(array)`, respectively.

An array is "strided" if it is stored in memory with well-defined spacings (strides) between its elements. A strided array with a supported element type may be passed to an external (non-Julia) library like BLAS or LAPACK by simply passing its `pointer` and the stride for each dimension. The `stride(A, d)` is the distance between elements along dimension `d`. For example, the builtin `Array` returned by `rand(5,7,2)` has its elements arranged contiguously in column major order. This means that the stride of the first dimension — the spacing between elements in the same column — is 1:

```
julia> A = rand(5, 7, 2);

julia> stride(A, 1)
1
```

The stride of the second dimension is the spacing between elements in the same row, skipping as many elements as there are in a single column (5). Similarly, jumping between the two "pages" (in the third

dimension) requires skipping 5*7 == 35 elements. The `strides` of this array is the tuple of these three numbers together:

```
julia> strides(A)
(1, 5, 35)
```

In this particular case, the number of elements skipped *in memory* matches the number of *linear indices* skipped. This is only the case for contiguous arrays like `Array` (and other `DenseArray` subtypes) and is not true in general. Views with range indices are a good example of *non-contiguous* strided arrays; consider `V = @view A[1:3:4, 2:2:6, 2:-1:1]`. This view `V` refers to the same memory as `A` but is skipping and re-arranging some of its elements. The stride of the first dimension of `V` is 3 because we're only selecting every third row from our original array:

```
julia> V = @view A[1:3:4, 2:2:6, 2:-1:1];

julia> stride(V, 1)
3
```

This view is similarly selecting every other column from our original `A` — and thus it needs to skip the equivalent of two five-element columns when moving between indices in the second dimension:

```
julia> stride(V, 2)
10
```

The third dimension is interesting because its order is reversed! Thus to get from the first "page" to the second one it must go *backwards* in memory, and so its stride in this dimension is negative!

```
julia> stride(V, 3)
-35
```

This means that the `pointer` for `V` is actually pointing into the middle of `A`'s memory block, and it refers to elements both backwards and forwards in memory. See the interface guide for strided arrays for more details on defining your own strided arrays. `StridedVector` and `StridedMatrix` are convenient aliases for many of the builtin array types that are considered strided arrays, allowing them to dispatch to select specialized implementations that call highly tuned and optimized BLAS and LAPACK functions using just the pointer and strides.

It is worth emphasizing that strides are about offsets in memory rather than indexing. If you are looking to convert between linear (single-index) indexing and cartesian (multi-index) indexing, see `LinearIndices` and `CartesianIndices`.

Chapter 21

Missing Values

Julia provides support for representing missing values in the statistical sense. This is for situations where no value is available for a variable in an observation, but a valid value theoretically exists. Missing values are represented via the `missing` object, which is the singleton instance of the type `Missing`. `missing` is equivalent to NULL in SQL and NA in R, and behaves like them in most situations.

21.1 Propagation of Missing Values

`missing` values *propagate* automatically when passed to standard mathematical operators and functions. For these functions, uncertainty about the value of one of the operands induces uncertainty about the result. In practice, this means a math operation involving a `missing` value generally returns `missing`:

```
julia> missing + 1
missing

julia> "a" * missing
missing

julia> abs(missing)
missing
```

Since `missing` is a normal Julia object, this propagation rule only works for functions which have opted in to implement this behavior. This can be achieved by:

- adding a specific method defined for arguments of type `Missing`,
- accepting arguments of this type, and passing them to functions which propagate them (like standard math operators).

Packages should consider whether it makes sense to propagate missing values when defining new functions, and define methods appropriately if this is the case. Passing a `missing` value to a function which does not have a method accepting arguments of type `Missing` throws a `MethodError`, just like for any other type.

Functions that do not propagate `missing` values can be made to do so by wrapping them in the `passmissing` function provided by the Missings.jl package. For example, `f(x)` becomes `passmissing(f)(x)`.

21.2 Equality and Comparison Operators

Standard equality and comparison operators follow the propagation rule presented above: if any of the operands is `missing`, the result is `missing`. Here are a few examples:

```
julia> missing == 1
missing

julia> missing == missing
missing

julia> missing < 1
missing

julia> 2 >= missing
missing
```

In particular, note that `missing == missing` returns `missing`, so `==` cannot be used to test whether a value is missing. To test whether `x` is `missing`, use `ismissing(x)`.

Special comparison operators `isequal` and `===` are exceptions to the propagation rule. They will always return a `Bool` value, even in the presence of `missing` values, considering `missing` as equal to `missing` and as different from any other value. They can therefore be used to test whether a value is `missing`:

```
julia> missing === 1
false

julia> isequal(missing, 1)
false

julia> missing === missing
true

julia> isequal(missing, missing)
true
```

The `isless` operator is another exception: `missing` is considered as greater than any other value. This operator is used by `sort!`, which therefore places `missing` values after all other values:

```
julia> isless(1, missing)
true

julia> isless(missing, Inf)
false

julia> isless(missing, missing)
false
```

21.3 Logical operators

Logical (or boolean) operators `|`, `&` and `xor` are another special case since they only propagate `missing` values when it is logically required. For these operators, whether or not the result is uncertain, depends on the particular operation. This follows the well-established rules of *three-valued logic* which are implemented by e.g. `NULL` in SQL and `NA` in R. This abstract definition corresponds to a relatively natural behavior which is best explained via concrete examples.

Let us illustrate this principle with the logical "or" operator `|`. Following the rules of boolean logic, if one of the operands is `true`, the value of the other operand does not have an influence on the result, which will always be `true`:

```
julia> true | true
true

julia> true | false
true

julia> false | true
true
```

Based on this observation, we can conclude if one of the operands is `true` and the other `missing`, we know that the result is `true` in spite of the uncertainty about the actual value of one of the operands. If we had been able to observe the actual value of the second operand, it could only be `true` or `false`, and in both cases the result would be `true`. Therefore, in this particular case, missingness does *not* propagate:

```
julia> true | missing
true

julia> missing | true
true
```

On the contrary, if one of the operands is `false`, the result could be either `true` or `false` depending on the value of the other operand. Therefore, if that operand is `missing`, the result has to be `missing` too:

```
julia> false | true
true

julia> true | false
true

julia> false | false
false

julia> false | missing
missing

julia> missing | false
missing
```

The behavior of the logical "and" operator `&` is similar to that of the `|` operator, with the difference that missingness does not propagate when one of the operands is `false`. For example, when that is the case of the first operand:

```
julia> false & false
false

julia> false & true
false

julia> false & missing
false
```

On the other hand, missingness propagates when one of the operands is `true`, for example the first one:

```
julia> true & true
true

julia> true & false
false

julia> true & missing
missing
```

Finally, the "exclusive or" logical operator `xor` always propagates `missing` values, since both operands always have an effect on the result. Also note that the negation operator `!` returns `missing` when the operand is `missing`, just like other unary operators.

21.4 Control Flow and Short-Circuiting Operators

Control flow operators including `if`, `while` and the ternary operator `x ? y : z` do not allow for missing values. This is because of the uncertainty about whether the actual value would be `true` or `false` if we could observe it. This implies we do not know how the program should behave. In this case, a `TypeError` is thrown as soon as a `missing` value is encountered in this context:

```
julia> if missing
           println("here")
       end
ERROR: TypeError: non-boolean (Missing) used in boolean context
```

For the same reason, contrary to logical operators presented above, the short-circuiting boolean operators `&&` and `||` do not allow for `missing` values in situations where the value of the operand determines whether the next operand is evaluated or not. For example:

```
julia> missing || false
ERROR: TypeError: non-boolean (Missing) used in boolean context

julia> missing && false
ERROR: TypeError: non-boolean (Missing) used in boolean context
```

```
julia> true && missing && false
ERROR: TypeError: non-boolean (Missing) used in boolean context
```

In contrast, there is no error thrown when the result can be determined without the `missing` values. This is the case when the code short-circuits before evaluating the `missing` operand, and when the `missing` operand is the last one:

```
julia> true && missing
missing

julia> false && missing
false
```

21.5 Arrays With Missing Values

Arrays containing missing values can be created like other arrays:

```
julia> [1, missing]
2-element Vector{Union{Missing, Int64}}:
 1
  missing
```

As this example shows, the element type of such arrays is `Union{Missing, T}`, with `T` the type of the non-missing values. This reflects the fact that array entries can be either of type `T` (here, `Int64`) or of type `Missing`. This kind of array uses an efficient memory storage equivalent to an `Array{T}` holding the actual values combined with an `Array{UInt8}` indicating the type of the entry (i.e. whether it is `Missing` or `T`).

Arrays allowing for missing values can be constructed with the standard syntax. Use `Array{Union{Missing, T}}(missing, dims)` to create arrays filled with missing values:

```
julia> Array{Union{Missing, String}}(missing, 2, 3)
2×3 Matrix{Union{Missing, String}}:
 missing  missing  missing
 missing  missing  missing
```

Note

Using `undef` or `similar` may currently give an array filled with `missing`, but this is not the correct way to obtain such an array. Use a `missing` constructor as shown above instead.

An array with element type allowing `missing` entries (e.g. `Vector{Union{Missing, T}}`) which does not contain any `missing` entries can be converted to an array type that does not allow for `missing` entries (e.g. `Vector{T}`) using `convert`. If the array contains `missing` values, a `MethodError` is thrown during conversion:

```
julia> x = Union{Missing, String}["a", "b"]
2-element Vector{Union{Missing, String}}:
 "a"
```

```
 "b"

julia> convert(Array{String}, x)
2-element Vector{String}:
 "a"
 "b"

julia> y = Union{Missing, String}[missing, "b"]
2-element Vector{Union{Missing, String}}:
 missing
 "b"

julia> convert(Array{String}, y)
ERROR: MethodError: Cannot `convert` an object of type Missing to an object of type String
```

21.6 Skipping Missing Values

Since `missing` values propagate with standard mathematical operators, reduction functions return `missing` when called on arrays which contain missing values:

```
julia> sum([1, missing])
missing
```

In this situation, use the `skipmissing` function to skip missing values:

```
julia> sum(skipmissing([1, missing]))
1
```

This convenience function returns an iterator which filters out `missing` values efficiently. It can therefore be used with any function which supports iterators:

```
julia> x = skipmissing([3, missing, 2, 1])
skipmissing(Union{Missing, Int64}[3, missing, 2, 1])

julia> maximum(x)
3

julia> sum(x)
6

julia> mapreduce(sqrt, +, x)
4.146264369941973
```

Objects created by calling `skipmissing` on an array can be indexed using indices from the parent array. Indices corresponding to missing values are not valid for these objects, and an error is thrown when trying to use them (they are also skipped by `keys` and `eachindex`):

```
julia> x[1]
3
```

```
julia> x[2]
ERROR: MissingException: the value at index (2,) is missing
[...]
```

This allows functions which operate on indices to work in combination with `skipmissing`. This is notably the case for search and find functions. These functions return indices valid for the object returned by `skipmissing`, and are also the indices of the matching entries *in the parent array*:

```
julia> findall(==(1), x)
1-element Vector{Int64}:
 4

julia> findfirst(!iszero, x)
1

julia> argmax(x)
1
```

Use `collect` to extract non-`missing` values and store them in an array:

```
julia> collect(x)
3-element Vector{Int64}:
 3
 2
 1
```

21.7 Logical Operations on Arrays

The three-valued logic described above for logical operators is also used by logical functions applied to arrays. Thus, array equality tests using the `==` operator return `missing` whenever the result cannot be determined without knowing the actual value of the `missing` entry. In practice, this means `missing` is returned if all non-missing values of the compared arrays are equal, but one or both arrays contain missing values (possibly at different positions):

```
julia> [1, missing] == [2, missing]
false

julia> [1, missing] == [1, missing]
missing

julia> [1, 2, missing] == [1, missing, 2]
missing
```

As for single values, use `isequal` to treat `missing` values as equal to other `missing` values, but different from non-missing values:

```
julia> isequal([1, missing], [1, missing])
true
```

```
julia> isequal([1, 2, missing], [1, missing, 2])
false
```

Functions `any` and `all` also follow the rules of three-valued logic. Thus, returning `missing` when the result cannot be determined:

```
julia> all([true, missing])
missing

julia> all([false, missing])
false

julia> any([true, missing])
true

julia> any([false, missing])
missing
```

Chapter 22

Networking and Streams

Julia provides a rich interface to deal with streaming I/O objects such as terminals, pipes and TCP sockets. These objects allow data to be sent and received in a stream-like fashion, which means that data is processed sequentially as it becomes available. This interface, though asynchronous at the system level, is presented in a synchronous manner to the programmer. This is achieved by making heavy use of Julia cooperative threading (coroutine) functionality.

22.1 Basic Stream I/O

All Julia streams expose at least a `read` and a `write` method, taking the stream as their first argument, e.g.:

```
julia> write(stdout, "Hello World");  # suppress return value 11 with ;
Hello World
julia> read(stdin, Char)

'\n': ASCII/Unicode U+000a (category Cc: Other, control)
```

Note that `write` returns 11, the number of bytes (in `"Hello World"`) written to `stdout`, but this return value is suppressed with the `;`.

Here Enter was pressed again so that Julia would read the newline. Now, as you can see from this example, `write` takes the data to write as its second argument, while `read` takes the type of the data to be read as the second argument.

For example, to read a simple byte array, we could do:

```
julia> x = zeros(UInt8, 4)
4-element Vector{UInt8}:
 0x00
 0x00
 0x00
 0x00

julia> read!(stdin, x)
abcd
4-element Vector{UInt8}:
```

```
0x61
0x62
0x63
0x64
```

However, since this is slightly cumbersome, there are several convenience methods provided. For example, we could have written the above as:

```
julia> read(stdin, 4)
abcd
4-element Vector{UInt8}:
 0x61
 0x62
 0x63
 0x64
```

or if we had wanted to read the entire line instead:

```
julia> readline(stdin)
abcd
"abcd"
```

Note that depending on your terminal settings, your TTY ("teletype terminal") may be line buffered and might thus require an additional enter before `stdin` data is sent to Julia. When running Julia from the command line in a TTY, output is sent to the console by default, and standard input is read from the keyboard.

To read every line from `stdin` you can use `eachline`:

```
for line in eachline(stdin)
    print("Found $line")
end
```

or `read` if you wanted to read by character instead:

```
while !eof(stdin)
    x = read(stdin, Char)
    println("Found: $x")
end
```

22.2 Text I/O

Note that the `write` method mentioned above operates on binary streams. In particular, values do not get converted to any canonical text representation but are written out as is:

```
julia> write(stdout, 0x61);  # suppress return value 1 with ;
a
```

Note that a is written to `stdout` by the `write` function and that the returned value is 1 (since `0x61` is one byte).

For text I/O, use the `print` or `show` methods, depending on your needs (see the documentation for these two methods for a detailed discussion of the difference between them):

```
julia> print(stdout, 0x61)
97
```

See Custom pretty-printing for more information on how to implement display methods for custom types.

22.3 IO Output Contextual Properties

Sometimes IO output can benefit from the ability to pass contextual information into show methods. The `IOContext` object provides this framework for associating arbitrary metadata with an IO object. For example, `:compact => true` adds a hinting parameter to the IO object that the invoked show method should print a shorter output (if applicable). See the `IOContext` documentation for a list of common properties.

22.4 Working with Files

You can write content to a file with the `write(filename::String, content)` method:

```
julia> write("hello.txt", "Hello, World!")
13
```

(13 is the number of bytes written.)

You can read the contents of a file with the `read(filename::String)` method, or `read(filename::String, String)` to the contents as a string:

```
julia> read("hello.txt", String)
"Hello, World!"
```

Advanced: streaming files

The `read` and `write` methods above allow you to read and write file contents. Like many other environments, Julia also has an `open` function, which takes a filename and returns an `IOStream` object that you can use to read and write things from the file. For example, if we have a file, `hello.txt`, whose contents are `Hello, World!`:

```
julia> f = open("hello.txt")
IOStream(<file hello.txt>)

julia> readlines(f)
1-element Vector{String}:
 "Hello, World!"
```

If you want to write to a file, you can open it with the write (`"w"`) flag:

```
julia> f = open("hello.txt","w")
IOStream(<file hello.txt>)

julia> write(f,"Hello again.")
12
```

If you examine the contents of `hello.txt` at this point, you will notice that it is empty; nothing has actually been written to disk yet. This is because the `IOStream` must be closed before the write is actually flushed to disk:

```
julia> close(f)
```

Examining `hello.txt` again will show its contents have been changed.

Opening a file, doing something to its contents, and closing it again is a very common pattern. To make this easier, there exists another invocation of `open` which takes a function as its first argument and filename as its second, opens the file, calls the function with the file as an argument, and then closes it again. For example, given a function:

```
function read_and_capitalize(f::IOStream)
    return uppercase(read(f, String))
end
```

You can call:

```
julia> open(read_and_capitalize, "hello.txt")
"HELLO AGAIN."
```

to open `hello.txt`, call `read_and_capitalize` on it, close `hello.txt` and return the capitalized contents.

To avoid even having to define a named function, you can use the `do` syntax, which creates an anonymous function on the fly:

```
julia> open("hello.txt") do f
           uppercase(read(f, String))
       end
"HELLO AGAIN."
```

If you want to redirect stdout to a file

```
out_file = open("output.txt", "w")

# Redirect stdout to file
redirect_stdout(out_file) do
    # Your code here
    println("This output goes to `out_file` via the `stdout` variable.")
end

# Close file
close(out_file)
```

Redirecting stdout to a file can help you save and analyze program output, automate processes, and meet compliance requirements.

22.5 A simple TCP example

Let's jump right in with a simple example involving TCP sockets. This functionality is in a standard library package called `Sockets`. Let's first create a simple server:

```
julia> using Sockets

julia> errormonitor(Threads.@spawn begin
           server = listen(2000)
           while true
               sock = accept(server)
               println("Hello World\n")
           end
       end)
Task (runnable) @0x00007fd31dc11ae0
```

To those familiar with the Unix socket API, the method names will feel familiar, though their usage is somewhat simpler than the raw Unix socket API. The first call to `listen` will create a server waiting for incoming connections on the specified port (2000) in this case. The same function may also be used to create various other kinds of servers:

```
julia> listen(2000) # Listens on localhost:2000 (IPv4)
Sockets.TCPServer(active)

julia> listen(ip"127.0.0.1",2000) # Equivalent to the first
Sockets.TCPServer(active)

julia> listen(ip"::1",2000) # Listens on localhost:2000 (IPv6)
Sockets.TCPServer(active)

julia> listen(IPv4(0),2001) # Listens on port 2001 on all IPv4 interfaces
Sockets.TCPServer(active)

julia> listen(IPv6(0),2001) # Listens on port 2001 on all IPv6 interfaces
Sockets.TCPServer(active)

julia> listen("testsocket") # Listens on a UNIX domain socket
Sockets.PipeServer(active)

julia> listen("\\\\.\\pipe\\testsocket") # Listens on a Windows named pipe
Sockets.PipeServer(active)
```

Note that the return type of the last invocation is different. This is because this server does not listen on TCP, but rather on a named pipe (Windows) or UNIX domain socket. Also note that Windows named pipe format has to be a specific pattern such that the name prefix (`\\.\pipe\`) uniquely identifies the file type. The difference between TCP and named pipes or UNIX domain sockets is subtle and has to do with the `accept` and `connect` methods. The `accept` method retrieves a connection to the client that is connecting

on the server we just created, while the `connect` function connects to a server using the specified method. The `connect` function takes the same arguments as `listen`, so, assuming the environment (i.e. host, cwd, etc.) is the same you should be able to pass the same arguments to `connect` as you did to listen to establish the connection. So let's try that out (after having created the server above):

```
julia> connect(2000)
TCPSocket(open, 0 bytes waiting)

julia> Hello World
```

As expected we saw "Hello World" printed. So, let's actually analyze what happened behind the scenes. When we called `connect`, we connect to the server we had just created. Meanwhile, the accept function returns a server-side connection to the newly created socket and prints "Hello World" to indicate that the connection was successful.

A great strength of Julia is that since the API is exposed synchronously even though the I/O is actually happening asynchronously, we didn't have to worry about callbacks or even making sure that the server gets to run. When we called `connect` the current task waited for the connection to be established and only continued executing after that was done. In this pause, the server task resumed execution (because a connection request was now available), accepted the connection, printed the message and waited for the next client. Reading and writing works in the same way. To see this, consider the following simple echo server:

```
julia> errormonitor(Threads.@spawn begin
           server = listen(2001)
           while true
               sock = accept(server)
               Threads.@spawn while isopen(sock)
                   write(sock, readline(sock, keep=true))
               end
           end
       end)
Task (runnable) @0x00007fd31dc12e60

julia> clientside = connect(2001)
TCPSocket(RawFD(28) open, 0 bytes waiting)

julia> errormonitor(Threads.@spawn while isopen(clientside)
           write(stdout, readline(clientside, keep=true))
       end)
Task (runnable) @0x00007fd31dc11870

julia> println(clientside,"Hello World from the Echo Server")
Hello World from the Echo Server
```

As with other streams, use `close` to disconnect the socket:

```
julia> close(clientside)
```

22.6 Resolving IP Addresses

One of the `connect` methods that does not follow the `listen` methods is `connect(host::String,port)`, which will attempt to connect to the host given by the `host` parameter on the port given by the `port` parameter. It allows you to do things like:

```
julia> connect("google.com", 80)
TCPSocket(RawFD(30) open, 0 bytes waiting)
```

At the base of this functionality is `getaddrinfo`, which will do the appropriate address resolution:

```
julia> getaddrinfo("google.com")
ip"74.125.226.225"
```

22.7 Asynchronous I/O

All I/O operations exposed by `Base.read` and `Base.write` can be performed asynchronously through the use of coroutines. You can create a new coroutine to read from or write to a stream using the `Threads.@spawn` macro:

```
julia> task = Threads.@spawn open("foo.txt", "w") do io
           write(io, "Hello, World!")
       end;

julia> wait(task)

julia> readlines("foo.txt")
1-element Vector{String}:
 "Hello, World!"
```

It's common to run into situations where you want to perform multiple asynchronous operations concurrently and wait until they've all completed. You can use the `@sync` macro to cause your program to block until all of the coroutines it wraps around have exited:

```
julia> using Sockets

julia> @sync for hostname in ("google.com", "github.com", "julialang.org")
           Threads.@spawn begin
               conn = connect(hostname, 80)
               write(conn, "GET / HTTP/1.1\r\nHost:$(hostname)\r\n\r\n")
               readline(conn, keep=true)
               println("Finished connection to $(hostname)")
           end
       end
Finished connection to google.com
Finished connection to julialang.org
Finished connection to github.com
```

22.8 Multicast

Julia supports multicast over IPv4 and IPv6 using the User Datagram Protocol (UDP) as transport.

Unlike the Transmission Control Protocol (TCP), UDP makes almost no assumptions about the needs of the application. TCP provides flow control (it accelerates and decelerates to maximize throughput), reliability (lost or corrupt packets are automatically retransmitted), sequencing (packets are ordered by the operating system before they are given to the application), segment size, and session setup and teardown. UDP provides no such features.

A common use for UDP is in multicast applications. TCP is a stateful protocol for communication between exactly two devices. UDP can use special multicast addresses to allow simultaneous communication between many devices.

Receiving IP Multicast Packets

To transmit data over UDP multicast, simply recv on the socket, and the first packet received will be returned. Note that it may not be the first packet that you sent however!

```
using Sockets
group = ip"228.5.6.7"
socket = Sockets.UDPSocket()
bind(socket, ip"0.0.0.0", 6789)
join_multicast_group(socket, group)
println(String(recv(socket)))
leave_multicast_group(socket, group)
close(socket)
```

Sending IP Multicast Packets

To transmit data over UDP multicast, simply send to the socket. Notice that it is not necessary for a sender to join the multicast group.

```
using Sockets
group = ip"228.5.6.7"
socket = Sockets.UDPSocket()
send(socket, group, 6789, "Hello over IPv4")
close(socket)
```

IPv6 Example

This example gives the same functionality as the previous program, but uses IPv6 as the network-layer protocol.

Listener:

```
using Sockets
group = Sockets.IPv6("ff05::5:6:7")
socket = Sockets.UDPSocket()
bind(socket, Sockets.IPv6("::"), 6789)
join_multicast_group(socket, group)
println(String(recv(socket)))
```

```
leave_multicast_group(socket, group)
close(socket)
```

Sender:

```
using Sockets
group = Sockets.IPv6("ff05::5:6:7")
socket = Sockets.UDPSocket()
send(socket, group, 6789, "Hello over IPv6")
close(socket)
```

Chapter 23

Parallel Computing

Julia supports these four categories of concurrent and parallel programming:

1. **Asynchronous "tasks", or coroutines**:

   Julia Tasks allow suspending and resuming computations for I/O, event handling, producer-consumer processes, and similar patterns. Tasks can synchronize through operations like `wait` and `fetch`, and communicate via `Channel`s. While strictly not parallel computing by themselves, Julia lets you schedule `Task`s on several threads.

2. **Multi-threading**:

   Julia's multi-threading provides the ability to schedule Tasks simultaneously on more than one thread or CPU core, sharing memory. This is usually the easiest way to get parallelism on one's PC or on a single large multi-core server. Julia's multi-threading is composable. When one multi-threaded function calls another multi-threaded function, Julia will schedule all the threads globally on available resources, without oversubscribing.

3. **Distributed computing**:

   Distributed computing runs multiple Julia processes with separate memory spaces. These can be on the same computer or multiple computers. The `Distributed` standard library provides the capability for remote execution of a Julia function. With this basic building block, it is possible to build many different kinds of distributed computing abstractions. Packages like `DistributedArrays.jl` are an example of such an abstraction. On the other hand, packages like `MPI.jl` and `Elemental.jl` provide access to the existing MPI ecosystem of libraries.

4. **GPU computing**:

   The Julia GPU compiler provides the ability to run Julia code natively on GPUs. There is a rich ecosystem of Julia packages that target GPUs. The JuliaGPU.org website provides a list of capabilities, supported GPUs, related packages and documentation.

Chapter 24

Asynchronous Programming

When a program needs to interact with the outside world, for example communicating with another machine over the internet, operations in the program may need to happen in an unpredictable order. Say your program needs to download a file. We would like to initiate the download operation, perform other operations while we wait for it to complete, and then resume the code that needs the downloaded file when it is available. This sort of scenario falls in the domain of asynchronous programming, sometimes also referred to as concurrent programming (since, conceptually, multiple things are happening at once).

To address these scenarios, Julia provides `Task`s (also known by several other names, such as symmetric coroutines, lightweight threads, cooperative multitasking, or one-shot continuations). When a piece of computing work (in practice, executing a particular function) is designated as a `Task`, it becomes possible to interrupt it by switching to another `Task`. The original `Task` can later be resumed, at which point it will pick up right where it left off. At first, this may seem similar to a function call. However there are two key differences. First, switching tasks does not use any space, so any number of task switches can occur without consuming the call stack. Second, switching among tasks can occur in any order, unlike function calls, where the called function must finish executing before control returns to the calling function.

24.1 Basic Task operations

You can think of a `Task` as a handle to a unit of computational work to be performed. It has a create-start-run-finish lifecycle. Tasks are created by calling the `Task` constructor on a 0-argument function to run, or using the `@task` macro:

```
julia> t = @task begin; sleep(5); println("done"); end
Task (runnable) @0x00007f13a40c0eb0
```

`@task x` is equivalent to `Task(()->x)`.

This task will wait for five seconds, and then print `done`. However, it has not started running yet. We can run it whenever we're ready by calling `schedule`:

```
julia> schedule(t);
```

If you try this in the REPL, you will see that `schedule` returns immediately. That is because it simply adds `t` to an internal queue of tasks to run. Then, the REPL will print the next prompt and wait for more input.

Waiting for keyboard input provides an opportunity for other tasks to run, so at that point `t` will start. `t` calls `sleep`, which sets a timer and stops execution. If other tasks have been scheduled, they could run then. After five seconds, the timer fires and restarts `t`, and you will see done printed. `t` is then finished.

The `wait` function blocks the calling task until some other task finishes. So for example if you type

```
julia> schedule(t); wait(t)
```

instead of only calling `schedule`, you will see a five second pause before the next input prompt appears. That is because the REPL is waiting for `t` to finish before proceeding.

It is common to want to create a task and schedule it right away, so the macro `Threads.@spawn` is provided for that purpose — `Threads.@spawn x` is equivalent to `task = @task x; task.sticky = false; schedule(task)`.

24.2 Communicating with Channels

In some problems, the various pieces of required work are not naturally related by function calls; there is no obvious "caller" or "callee" among the jobs that need to be done. An example is the producer-consumer problem, where one complex procedure is generating values and another complex procedure is consuming them. The consumer cannot simply call a producer function to get a value, because the producer may have more values to generate and so might not yet be ready to return. With tasks, the producer and consumer can both run as long as they need to, passing values back and forth as necessary.

Julia provides a `Channel` mechanism for solving this problem. A `Channel` is a waitable first-in first-out queue which can have multiple tasks reading from and writing to it.

Let's define a producer task, which produces values via the `put!` call. To consume values, we need to schedule the producer to run in a new task. A special `Channel` constructor which accepts a 1-arg function as an argument can be used to run a task bound to a channel. We can then `take!` values repeatedly from the channel object:

```
julia> function producer(c::Channel)
           put!(c, "start")
           for n=1:4
               put!(c, 2n)
           end
           put!(c, "stop")
       end;

julia> chnl = Channel(producer);

julia> take!(chnl)
"start"

julia> take!(chnl)
2

julia> take!(chnl)
4

julia> take!(chnl)
```

```
6

julia> take!(chnl)
8

julia> take!(chnl)
"stop"
```

One way to think of this behavior is that `producer` was able to return multiple times. Between calls to `put!`, the producer's execution is suspended and the consumer has control.

The returned `Channel` can be used as an iterable object in a `for` loop, in which case the loop variable takes on all the produced values. The loop is terminated when the channel is closed.

```
julia> for x in Channel(producer)
           println(x)
       end
start
2
4
6
8
stop
```

Note that we did not have to explicitly close the channel in the producer. This is because the act of binding a `Channel` to a `Task` associates the open lifetime of a channel with that of the bound task. The channel object is closed automatically when the task terminates. Multiple channels can be bound to a task, and vice-versa.

While the `Task` constructor expects a 0-argument function, the `Channel` method that creates a task-bound channel expects a function that accepts a single argument of type `Channel`. A common pattern is for the producer to be parameterized, in which case a partial function application is needed to create a 0 or 1 argument anonymous function.

For `Task` objects this can be done either directly or by use of a convenience macro:

```
function mytask(myarg)
    ...
end

taskHdl = Task(() -> mytask(7))
# or, equivalently
taskHdl = @task mytask(7)
```

To orchestrate more advanced work distribution patterns, `bind` and `schedule` can be used in conjunction with `Task` and `Channel` constructors to explicitly link a set of channels with a set of producer/consumer tasks.

More on Channels

A channel can be visualized as a pipe, i.e., it has a write end and a read end:

- Multiple writers in different tasks can write to the same channel concurrently via `put!` calls.
- Multiple readers in different tasks can read data concurrently via `take!` calls.
- As an example:

```
# Given Channels c1 and c2,
c1 = Channel(32)
c2 = Channel(32)

# and a function `foo` which reads items from c1, processes the item read
# and writes a result to c2,
function foo()
    while true
        data = take!(c1)
        [...]               # process data
        put!(c2, result)    # write out result
    end
end

# we can schedule `n` instances of `foo` to be active concurrently.
for _ in 1:n
    errormonitor(Threads.@spawn foo())
end
```

- Channels are created via the `Channel{T}(sz)` constructor. The channel will only hold objects of type T. If the type is not specified, the channel can hold objects of any type. `sz` refers to the maximum number of elements that can be held in the channel at any time. For example, `Channel(32)` creates a channel that can hold a maximum of 32 objects of any type. A `Channel{MyType}(64)` can hold up to 64 objects of `MyType` at any time.
- If a `Channel` is empty, readers (on a `take!` call) will block until data is available (see `isempty`).
- If a `Channel` is full, writers (on a `put!` call) will block until space becomes available (see `isfull`).
- `isready` tests for the presence of any object in the channel, while `wait` waits for an object to become available.
- Note that if another task is currently waiting to `put!` an object into a channel, a channel can have more items available than its capacity.
- A `Channel` is in an open state initially. This means that it can be read from and written to freely via `take!` and `put!` calls. `close` closes a `Channel`. On a closed `Channel`, `put!` will fail. For example:

```
julia> c = Channel(2);

julia> put!(c, 1) # `put!` on an open channel succeeds
1

julia> close(c);
```

```
julia> put!(c, 2) # `put!` on a closed channel throws an exception.
ERROR: InvalidStateException: Channel is closed.
Stacktrace:
[...]
```

- `take!` and `fetch` (which retrieves but does not remove the value) on a closed channel successfully return any existing values until it is emptied. Continuing the above example:

```
julia> fetch(c) # Any number of `fetch` calls succeed.
1

julia> fetch(c)
1

julia> take!(c) # The first `take!` removes the value.
1

julia> take!(c) # No more data available on a closed channel.
ERROR: InvalidStateException: Channel is closed.
Stacktrace:
[...]
```

Consider a simple example using channels for inter-task communication. We start 4 tasks to process data from a single `jobs` channel. Jobs, identified by an id (`job_id`), are written to the channel. Each task in this simulation reads a `job_id`, waits for a random amount of time and writes back a tuple of `job_id` and the simulated time to the results channel. Finally all the `results` are printed out.

```
julia> const jobs = Channel{Int}(32);

julia> const results = Channel{Tuple}(32);

julia> function do_work()
           for job_id in jobs
               exec_time = rand()
               sleep(exec_time)                # simulates elapsed time doing actual work
                                               # typically performed externally.
               put!(results, (job_id, exec_time))
           end
       end;

julia> function make_jobs(n)
           for i in 1:n
               put!(jobs, i)
           end
       end;

julia> n = 12;

julia> errormonitor(Threads.@spawn make_jobs(n)); # feed the jobs channel with "n" jobs

julia> for i in 1:4 # start 4 tasks to process requests in parallel
```

```
            errormonitor(Threads.@spawn do_work())
        end

julia> @elapsed while n > 0 # print out results
            job_id, exec_time = take!(results)
            println("$job_id finished in $(round(exec_time; digits=2)) seconds")
            global n = n - 1
        end
4 finished in 0.22 seconds
3 finished in 0.45 seconds
1 finished in 0.5 seconds
7 finished in 0.14 seconds
2 finished in 0.78 seconds
5 finished in 0.9 seconds
9 finished in 0.36 seconds
6 finished in 0.87 seconds
8 finished in 0.79 seconds
10 finished in 0.64 seconds
12 finished in 0.5 seconds
11 finished in 0.97 seconds
0.029772311
```

Instead of `errormonitor(t)`, a more robust solution may be to use `bind(results, t)`, as that will not only log any unexpected failures, but also force the associated resources to close and propagate the exception everywhere.

24.3 More task operations

Task operations are built on a low-level primitive called `yieldto`. `yieldto(task, value)` suspends the current task, switches to the specified `task`, and causes that task's last `yieldto` call to return the specified `value`. Notice that `yieldto` is the only operation required to use task-style control flow; instead of calling and returning we are always just switching to a different task. This is why this feature is also called "symmetric coroutines"; each task is switched to and from using the same mechanism.

`yieldto` is powerful, but most uses of tasks do not invoke it directly. Consider why this might be. If you switch away from the current task, you will probably want to switch back to it at some point, but knowing when to switch back, and knowing which task has the responsibility of switching back, can require considerable coordination. For example, `put!` and `take!` are blocking operations, which, when used in the context of channels maintain state to remember who the consumers are. Not needing to manually keep track of the consuming task is what makes `put!` easier to use than the low-level `yieldto`.

In addition to `yieldto`, a few other basic functions are needed to use tasks effectively.

- `current_task` gets a reference to the currently-running task.
- `istaskdone` queries whether a task has exited.
- `istaskstarted` queries whether a task has run yet.
- `task_local_storage` manipulates a key-value store specific to the current task.

24.4 Tasks and events

Most task switches occur as a result of waiting for events such as I/O requests, and are performed by a scheduler included in Julia Base. The scheduler maintains a queue of runnable tasks, and executes an event loop that restarts tasks based on external events such as message arrival.

The basic function for waiting for an event is `wait`. Several objects implement `wait`; for example, given a `Process` object, `wait` will wait for it to exit. `wait` is often implicit; for example, a `wait` can happen inside a call to `read` to wait for data to be available.

In all of these cases, `wait` ultimately operates on a `Condition` object, which is in charge of queueing and restarting tasks. When a task calls `wait` on a `Condition`, the task is marked as non-runnable, added to the condition's queue, and switches to the scheduler. The scheduler will then pick another task to run, or block waiting for external events. If all goes well, eventually an event handler will call `notify` on the condition, which causes tasks waiting for that condition to become runnable again.

A task created explicitly by calling `Task` is initially not known to the scheduler. This allows you to manage tasks manually using `yieldto` if you wish. However, when such a task waits for an event, it still gets restarted automatically when the event happens, as you would expect.

Chapter 25

Multi-Threading

Visit this blog post for a presentation of Julia multi-threading features.

25.1 Starting Julia with multiple threads

By default, Julia starts up with 2 threads of execution; 1 worker thread and 1 interactive thread. This can be verified by using the command `Threads.nthreads()`:

```
julia> Threads.nthreads(:default)
1
julia> Threads.nthreads(:interactive)
1
```

The number of execution threads is controlled either by using the `-t`/`--threads` command line argument or by using the `JULIA_NUM_THREADS` environment variable. When both are specified, then `-t`/`--threads` takes precedence.

The number of threads can either be specified as an integer (`--threads=4`) or as `auto` (`--threads=auto`), where `auto` tries to infer a useful default number of threads to use (see Command-line Options for more details).

See threadpools for how to control how many `:default` and `:interactive` threads are in each threadpool.

Julia 1.5

The `-t`/`--threads` command line argument requires at least Julia 1.5. In older versions you must use the environment variable instead.

Julia 1.7

Using `auto` as value of the environment variable `JULIA_NUM_THREADS` requires at least Julia 1.7. In older versions, this value is ignored.

Julia 1.12

Starting by default with 1 interactive thread, as well as the 1 worker thread, was made as such in Julia 1.12 If the number of threads is set to 1 by either doing `-t1` or `JULIA_NUM_THREADS=1` an interactive thread will not be spawned.

Lets start Julia with 4 threads:

```
$ julia --threads 4
```

Let's verify there are 4 threads at our disposal.

```
julia> Threads.nthreads()
4
```

But we are currently on the master thread. To check, we use the function `Threads.threadid`

```
julia> Threads.threadid()
1
```

Note

If you prefer to use the environment variable you can set it as follows in Bash (Linux/macOS):

```
export JULIA_NUM_THREADS=4
```

C shell on Linux/macOS, CMD on Windows:

```
set JULIA_NUM_THREADS=4
```

Powershell on Windows:

```
$env:JULIA_NUM_THREADS=4
```

Note that this must be done *before* starting Julia.

Note

The number of threads specified with `-t/--threads` is propagated to worker processes that are spawned using the `-p/--procs` or `--machine-file` command line options. For example, `julia -p2 -t2` spawns 1 main process with 2 worker processes, and all three processes have 2 threads enabled. For more fine grained control over worker threads use `addprocs` and pass `-t/--threads` as `exeflags`.

Multiple GC Threads

The Garbage Collector (GC) can use multiple threads. The amount used by default matches the compute worker threads or can configured by either the `--gcthreads` command line argument or by using the `JULIA_NUM_GC_THREADS` environment variable.

Julia 1.10

The `--gcthreads` command line argument requires at least Julia 1.10.

For more details about garbage collection configuration and performance tuning, see Memory Management and Garbage Collection.

25.2 Threadpools

When a program's threads are busy with many tasks to run, tasks may experience delays which may negatively affect the responsiveness and interactivity of the program. To address this, you can specify that a task is interactive when you `Threads.@spawn` it:

```
using Base.Threads
@spawn :interactive f()
```

Interactive tasks should avoid performing high latency operations, and if they are long duration tasks, should yield frequently.

By default Julia starts with one interactive thread reserved to run interactive tasks, but that number can be controlled with:

```
$ julia --threads 3,1
julia> Threads.nthreads(:interactive)
1

$ julia --threads 3,0
julia> Threads.nthreads(:interactive)
0
```

The environment variable `JULIA_NUM_THREADS` can also be used similarly:

```
export JULIA_NUM_THREADS=3,1
```

This starts Julia with 3 threads in the `:default` threadpool and 1 thread in the `:interactive` threadpool:

```
julia> using Base.Threads

julia> nthreadpools()
2

julia> threadpool() # the main thread is in the interactive thread pool
:interactive
```

```
julia> nthreads(:default)
3

julia> nthreads(:interactive)
1

julia> nthreads()
3
```

Note

Explicitly asking for 1 thread by doing `-t1` or `JULIA_NUM_THREADS=1` does not add an interactive thread.

Note

The zero-argument version of `nthreads` returns the number of threads in the default pool.

Note

Depending on whether Julia has been started with interactive threads, the main thread is either in the default or interactive thread pool.

Either or both numbers can be replaced with the word `auto`, which causes Julia to choose a reasonable default.

25.3 The `@threads` Macro

Let's work a simple example using our native threads. Let us create an array of zeros:

```
julia> a = zeros(10)
10-element Vector{Float64}:
 0.0
 0.0
 0.0
 0.0
 0.0
 0.0
 0.0
 0.0
 0.0
 0.0
```

Let us operate on this array simultaneously using 4 threads. We'll have each thread write its thread ID into each location.

Julia supports parallel loops using the `Threads.@threads` macro. This macro is affixed in front of a `for` loop to indicate to Julia that the loop is a multi-threaded region:

```
julia> Threads.@threads for i = 1:10
```

```
            a[i] = Threads.threadid()
        end
```

The iteration space is split among the threads, after which each thread writes its thread ID to its assigned locations:

```
julia> a
10-element Vector{Float64}:
 1.0
 1.0
 1.0
 2.0
 2.0
 2.0
 3.0
 3.0
 4.0
 4.0
```

Note that `Threads.@threads` does not have an optional reduction parameter like `@distributed`.

Using `@threads` without data-races

The concept of a data-race is elaborated on in "Communication and data races between threads". For now, just know that a data race can result in incorrect results and dangerous errors.

Lets say we want to make the function `sum_single` below multithreaded.

```
julia> function sum_single(a)
            s = 0
            for i in a
                s += i
            end
            s
        end
sum_single (generic function with 1 method)

julia> sum_single(1:1_000_000)
500000500000
```

Simply adding `@threads` exposes a data race with multiple threads reading and writing `s` at the same time.

```
julia> function sum_multi_bad(a)
            s = 0
            Threads.@threads for i in a
                s += i
            end
            s
        end
sum_multi_bad (generic function with 1 method)
```

```
julia> sum_multi_bad(1:1_000_000)
70140554652
```

Note that the result is not `500000500000` as it should be, and will most likely change each evaluation.

To fix this, buffers that are specific to the task may be used to segment the sum into chunks that are race-free. Here `sum_single` is reused, with its own internal buffer `s`. The input vector `a` is split into at most `nthreads()` chunks for parallel work. We then use `Threads.@spawn` to create tasks that individually sum each chunk. Finally, we sum the results from each task using `sum_single` again:

```
julia> function sum_multi_good(a)
           chunks = Iterators.partition(a, cld(length(a), Threads.nthreads()))
           tasks = map(chunks) do chunk
               Threads.@spawn sum_single(chunk)
           end
           chunk_sums = fetch.(tasks)
           return sum_single(chunk_sums)
       end
sum_multi_good (generic function with 1 method)

julia> sum_multi_good(1:1_000_000)
500000500000
```

Note

Buffers should not be managed based on `threadid()` i.e. `buffers = zeros(Threads.nthreads())` because concurrent tasks can yield, meaning multiple concurrent tasks may use the same buffer on a given thread, introducing risk of data races. Further, when more than one thread is available tasks may change thread at yield points, which is known as task migration.

Another option is the use of atomic operations on variables shared across tasks/threads, which may be more performant depending on the characteristics of the operations.

25.4 Communication and data-races between threads

Although Julia's threads can communicate through shared memory, it is notoriously difficult to write correct and data-race free multi-threaded code. Julia's `Channel`s are thread-safe and may be used to communicate safely. There are also sections below that explain how to use locks and atomics to avoid data-races.

In certain cases, Julia is able to detect a detect safety violations, in particular in regards to deadlocks or other known-unsafe operations such as yielding to the currently running task. In these cases, a `ConcurrencyViolationError` is thrown.

Data-race freedom

You are entirely responsible for ensuring that your program is data-race free, and nothing promised here can be assumed if you do not observe that requirement. The observed results may be highly unintuitive.

If data-races are introduced, Julia is not memory safe. **Be very careful about reading *any* data if another thread might write to it, as it could result in segmentation faults or worse**. Below are a couple of unsafe ways to access global variables from different threads:

```
Thread 1:
global b = false
global a = rand()
global b = true

Thread 2:
while !b; end
bad_read1(a) # it is NOT safe to access `a` here!

Thread 3:
while !@isdefined(a); end
bad_read2(a) # it is NOT safe to access `a` here
```

Using locks to avoid data-races

An important tool for avoiding data races, and writing thread-safe code in general, is the concept of a "lock". A lock can be locked and unlocked. If a thread has locked a lock, and not unlocked it, it is said to "hold" the lock. If there is only one lock, and we write code that requires holding the lock to access some data, we can ensure that multiple threads will never access the same data simultaneously.

Note that the link between a lock and a variable is made by the programmer, and not the program. A helper-type `Base.Lockable` exists that helps you associate a lock and a value. This is often more safe than keeping track yourself, and is detailed under Using Base.Lockable to associate a lock and a value.

For example, we can create a lock `my_lock`, and lock it while we mutate a variable `my_variable`. This is done most simply with the `@lock` macro:

```
julia> my_lock = ReentrantLock();

julia> my_variable = [1, 2, 3];

julia> @lock my_lock my_variable[1] = 100
100
```

By using a similar pattern with the same lock and variable, but on another thread, the operations are free from data-races.

We could have performed the operation above with the functional version of `lock`, in the following two ways:

```
julia> lock(my_lock) do
           my_variable[1] = 100
       end
100

julia> begin
           lock(my_lock)
           try
               my_variable[1] = 100
           finally
               unlock(my_lock)
           end
```

```
      end
100
```

All three options are equivalent. Note how the final version requires an explicit `try`-block to ensure that the lock is always unlocked, whereas the first two version do this internally. One should always use the lock pattern above when changing data (such as assigning to a global or closure variable) accessed by other threads. Failing to do this could have unforeseen and serious consequences.

Using Base.Lockable to associate a lock and a value

As mentioned in the previous section, the helper-type `Base.Lockable` can be used to programmatically ensure the association between a lock and a value. This is generally recommended, as it is both less prone to error and more readable for others compared to having the association only by convention.

Any object can be wrapped in `Base.Lockable`:

```
julia> my_array = [];

julia> my_locked_array = Base.Lockable(my_array);
```

If the lock is held, the underlying object can be accessed with the empty indexing notation:

```
julia> begin
           lock(my_locked_array)
           try
               push!(my_locked_array[], 1)
           finally
               unlock(my_locked_array)
           end
       end
1-element Vector{Any}:
 1
```

It is usually easier and safer to pass a function as the first argument to `lock`. The function is applied to the unlocked object, and the locking/unlocking is handled automatically:

```
julia> lock(x -> push!(x, 2), my_locked_array);

julia> lock(display, my_locked_array)
2-element Vector{Any}:
 1
 2

julia> lock(my_locked_array) do x
           x[1] = π
           display(x)
       end
2-element Vector{Any}:
 π = 3.1415926535897...
 2
```

Atomic Operations

Julia supports accessing and modifying values *atomically*, that is, in a thread-safe way to avoid race conditions. A value (which must be of a primitive type) can be wrapped as `Threads.Atomic` to indicate it must be accessed in this way. Here we can see an example:

```
julia> i = Threads.Atomic{Int}(0);

julia> ids = zeros(4);

julia> old_is = zeros(4);

julia> Threads.@threads for id in 1:4
           old_is[id] = Threads.atomic_add!(i, id)
           ids[id] = id
       end

julia> old_is
4-element Vector{Float64}:
 0.0
 1.0
 7.0
 3.0

julia> i[]
 10

julia> ids
4-element Vector{Float64}:
 1.0
 2.0
 3.0
 4.0
```

Had we tried to do the addition without the atomic tag, we might have gotten the wrong answer due to a race condition. An example of what would happen if we didn't avoid the race:

```
julia> using Base.Threads

julia> Threads.nthreads()
4

julia> acc = Ref(0)
Base.RefValue{Int64}(0)

julia> @threads for i in 1:1000
          acc[] += 1
       end

julia> acc[]
926
```

```
julia> acc = Atomic{Int64}(0)
Atomic{Int64}(0)

julia> @threads for i in 1:1000
          atomic_add!(acc, 1)
       end

julia> acc[]
1000
```

Per-field atomics

We can also use atomics on a more granular level using the `@atomic`, `@atomicswap`, `@atomicreplace` macros, and `@atomiconce` macros.

Specific details of the memory model and other details of the design are written in the Julia Atomics Manifesto, which will later be published formally.

Any field in a struct declaration can be decorated with `@atomic`, and then any write must be marked with `@atomic` also, and must use one of the defined atomic orderings (`:monotonic`, `:acquire`, `:release`, `:acquire_release`, or `:sequentially_consistent`). Any read of an atomic field can also be annotated with an atomic ordering constraint, or will be done with monotonic (relaxed) ordering if unspecified.

Julia 1.7

Per-field atomics requires at least Julia 1.7.

25.5 Side effects and mutable function arguments

When using multi-threading we have to be careful when using functions that are not pure as we might get a wrong answer. For instance functions that have a name ending with `!` by convention modify their arguments and thus are not pure.

25.6 @threadcall

External libraries, such as those called via `ccall`, pose a problem for Julia's task-based I/O mechanism. If a C library performs a blocking operation, that prevents the Julia scheduler from executing any other tasks until the call returns. (Exceptions are calls into custom C code that call back into Julia, which may then yield, or C code that calls `jl_yield()`, the C equivalent of `yield`.)

The `@threadcall` macro provides a way to avoid stalling execution in such a scenario. It schedules a C function for execution in a separate thread. A threadpool with a default size of 4 is used for this. The size of the threadpool is controlled via environment variable `UV_THREADPOOL_SIZE`. While waiting for a free thread, and during function execution once a thread is available, the requesting task (on the main Julia event loop) yields to other tasks. Note that `@threadcall` does not return until the execution is complete. From a user point of view, it is therefore a blocking call like other Julia APIs.

It is very important that the called function does not call back into Julia, as it will segfault.

`@threadcall` may be removed/changed in future versions of Julia.

25.7 Caveats

At this time, most operations in the Julia runtime and standard libraries can be used in a thread-safe manner, if the user code is data-race free. However, in some areas work on stabilizing thread support is ongoing. Multi-threaded programming has many inherent difficulties, and if a program using threads exhibits unusual or undesirable behavior (e.g. crashes or mysterious results), thread interactions should typically be suspected first.

There are a few specific limitations and warnings to be aware of when using threads in Julia:

- Base collection types require manual locking if used simultaneously by multiple threads where at least one thread modifies the collection (common examples include `push!` on arrays, or inserting items into a `Dict`).
- The schedule used by `@spawn` is nondeterministic and should not be relied on.
- Compute-bound, non-memory-allocating tasks can prevent garbage collection from running in other threads that are allocating memory. In these cases it may be necessary to insert a manual call to `GC.safepoint()` to allow GC to run. This limitation will be removed in the future.
- Avoid running top-level operations, e.g. `include`, or `eval` of type, method, and module definitions in parallel.
- Be aware that finalizers registered by a library may break if threads are enabled. This may require some transitional work across the ecosystem before threading can be widely adopted with confidence. See the section on the safe use of finalizers for further details.

25.8 Task Migration

After a task starts running on a certain thread it may move to a different thread if the task yields.

Such tasks may have been started with `@spawn` or `@threads`, although the `:static` schedule option for `@threads` does freeze the threadid.

This means that in most cases `threadid()` should not be treated as constant within a task, and therefore should not be used to index into a vector of buffers or stateful objects.

Julia 1.7

Task migration was introduced in Julia 1.7. Before this tasks always remained on the same thread that they were started on.

25.9 Safe use of Finalizers

Because finalizers can interrupt any code, they must be very careful in how they interact with any global state. Unfortunately, the main reason that finalizers are used is to update global state (a pure function is generally rather pointless as a finalizer). This leads us to a bit of a conundrum. There are a few approaches to dealing with this problem:

1. When single-threaded, code could call the internal `jl_gc_enable_finalizers` C function to prevent finalizers from being scheduled inside a critical region. Internally, this is used inside some functions (such as our C locks) to prevent recursion when doing certain operations (incremental package loading, codegen, etc.). The combination of a lock and this flag can be used to make finalizers safe.

2. A second strategy, employed by Base in a couple places, is to explicitly delay a finalizer until it may be able to acquire its lock non-recursively. The following example demonstrates how this strategy could be applied to `Distributed.finalize_ref`:

```
function finalize_ref(r::AbstractRemoteRef)
    if r.where > 0 # Check if the finalizer is already run
        if islocked(client_refs) || !trylock(client_refs)
            # delay finalizer for later if we aren't free to acquire the lock
            finalizer(finalize_ref, r)
            return nothing
        end
        try # `lock` should always be followed by `try`
            if r.where > 0 # Must check again here
                # Do actual cleanup here
                r.where = 0
            end
        finally
            unlock(client_refs)
        end
    end
    nothing
end
```

3. A related third strategy is to use a yield-free queue. We don't currently have a lock-free queue implemented in Base, but `Base.IntrusiveLinkedListSynchronized{T}` is suitable. This can frequently be a good strategy to use for code with event loops. For example, this strategy is employed by `Gtk.jl` to manage lifetime ref-counting. In this approach, we don't do any explicit work inside the `finalizer`, and instead add it to a queue to run at a safer time. In fact, Julia's task scheduler already uses this, so defining the finalizer as `x -> @spawn do_cleanup(x)` is one example of this approach. Note however that this doesn't control which thread `do_cleanup` runs on, so `do_cleanup` would still need to acquire a lock. That doesn't need to be true if you implement your own queue, as you can explicitly only drain that queue from your thread.

Chapter 26

Multi-processing and Distributed Computing

An implementation of distributed memory parallel computing is provided by module `Distributed` as part of the standard library shipped with Julia.

Most modern computers possess more than one CPU, and several computers can be combined together in a cluster. Harnessing the power of these multiple CPUs allows many computations to be completed more quickly. There are two major factors that influence performance: the speed of the CPUs themselves, and the speed of their access to memory. In a cluster, it's fairly obvious that a given CPU will have fastest access to the RAM within the same computer (node). Perhaps more surprisingly, similar issues are relevant on a typical multicore laptop, due to differences in the speed of main memory and the `cache`. Consequently, a good multiprocessing environment should allow control over the "ownership" of a chunk of memory by a particular CPU. Julia provides a multiprocessing environment based on message passing to allow programs to run on multiple processes in separate memory domains at once.

Julia's implementation of message passing is different from other environments such as MPI[1]. Communication in Julia is generally "one-sided", meaning that the programmer needs to explicitly manage only one process in a two-process operation. Furthermore, these operations typically do not look like "message send" and "message receive" but rather resemble higher-level operations like calls to user functions.

Distributed programming in Julia is built on two primitives: *remote references* and *remote calls*. A remote reference is an object that can be used from any process to refer to an object stored on a particular process. A remote call is a request by one process to call a certain function on certain arguments on another (possibly the same) process.

Remote references come in two flavors: `Future` and `RemoteChannel`.

A remote call returns a `Future` to its result. Remote calls return immediately; the process that made the call proceeds to its next operation while the remote call happens somewhere else. You can wait for a remote call to finish by calling `wait` on the returned `Future`, and you can obtain the full value of the result using `fetch`.

On the other hand, `RemoteChannel` s are rewritable. For example, multiple processes can coordinate their processing by referencing the same remote `Channel`.

Each process has an associated identifier. The process providing the interactive Julia prompt always has an `id` equal to 1. The processes used by default for parallel operations are referred to as "workers". When there is only one process, process 1 is considered a worker. Otherwise, workers are considered to be all processes other than process 1. As a result, adding 2 or more processes is required to gain benefits from parallel processing methods like `pmap`. Adding a single process is beneficial if you just wish to do other things in the main process while a long computation is running on the worker.

Let's try this out. Starting with `julia -p n` provides `n` worker processes on the local machine. Generally it makes sense for `n` to equal the number of CPU threads (logical cores) on the machine. Note that the `-p` argument implicitly loads module `Distributed`.

```
$ julia -p 2

julia> r = remotecall(rand, 2, 2, 2)
Future(2, 1, 4, nothing)

julia> s = @spawnat 2 1 .+ fetch(r)
Future(2, 1, 5, nothing)

julia> fetch(s)
2×2 Matrix{Float64}:
 1.18526  1.50912
 1.16296  1.60607
```

The first argument to `remotecall` is the function to call. Most parallel programming in Julia does not reference specific processes or the number of processes available, but `remotecall` is considered a low-level interface providing finer control. The second argument to `remotecall` is the `id` of the process that will do the work, and the remaining arguments will be passed to the function being called.

As you can see, in the first line we asked process 2 to construct a 2-by-2 random matrix, and in the second line we asked it to add 1 to it. The result of both calculations is available in the two futures, `r` and `s`. The `@spawnat` macro evaluates the expression in the second argument on the process specified by the first argument.

Occasionally you might want a remotely-computed value immediately. This typically happens when you read from a remote object to obtain data needed by the next local operation. The function `remotecall_fetch` exists for this purpose. It is equivalent to `fetch(remotecall(...))` but is more efficient.

```
julia> remotecall_fetch(r-> fetch(r)[1, 1], 2, r)
0.18526337335308085
```

This fetches the array on worker 2 and returns the first value. Note, that `fetch` doesn't move any data in this case, since it's executed on the worker that owns the array. One can also write:

```
julia> remotecall_fetch(getindex, 2, r, 1, 1)
0.10824216411304866
```

Remember that `getindex(r,1,1)` is equivalent to `r[1,1]`, so this call fetches the first element of the future `r`.

To make things easier, the symbol `:any` can be passed to `@spawnat`, which picks where to do the operation for you:

```
julia> r = @spawnat :any rand(2,2)
Future(2, 1, 4, nothing)

julia> s = @spawnat :any 1 .+ fetch(r)
```

```
Future(3, 1, 5, nothing)

julia> fetch(s)
2×2 Matrix{Float64}:
 1.38854  1.9098
 1.20939  1.57158
```

Note that we used `1 .+ fetch(r)` instead of `1 .+ r`. This is because we do not know where the code will run, so in general a `fetch` might be required to move `r` to the process doing the addition. In this case, `@spawnat` is smart enough to perform the computation on the process that owns `r`, so the `fetch` will be a no-op (no work is done).

(It is worth noting that `@spawnat` is not built-in but defined in Julia as a macro. It is possible to define your own such constructs.)

An important thing to remember is that, once fetched, a `Future` will cache its value locally. Further `fetch` calls do not entail a network hop. Once all referencing `Future`s have fetched, the remote stored value is deleted.

`Threads.@spawn` is similar to `@spawnat`, but only runs tasks on the local process. We use it to create a "feeder" task for each process. Each task picks the next index that needs to be computed, then waits for its process to finish, then repeats until we run out of indices. Note that the feeder tasks do not begin to execute until the main task reaches the end of the `@sync` block, at which point it surrenders control and waits for all the local tasks to complete before returning from the function. As for v0.7 and beyond, the feeder tasks are able to share state via `nextidx` because they all run on the same process. Even if `Task`s are scheduled cooperatively, locking may still be required in some contexts, as in asynchronous I/O. This means context switches only occur at well-defined points: in this case, when `remotecall_fetch` is called. This is the current state of implementation and it may change for future Julia versions, as it is intended to make it possible to run up to N `Task`s on M `Process`, aka M:N Threading. Then a lock acquiring\releasing model for `nextidx` will be needed, as it is not safe to let multiple processes read-write a resource at the same time.

26.1 Code Availability and Loading Packages

Your code must be available on any process that runs it. For example, type the following into the Julia prompt:

```
julia> function rand2(dims...)
           return 2*rand(dims...)
       end

julia> rand2(2,2)
2×2 Matrix{Float64}:
 0.153756  0.368514
 1.15119   0.918912

julia> fetch(@spawnat :any rand2(2,2))
ERROR: RemoteException(2, CapturedException(UndefVarError(Symbol("#rand2"))))
Stacktrace:
[...]
```

Process 1 knew about the function rand2, but process 2 did not.

Most commonly you'll be loading code from files or packages, and you have a considerable amount of flexibility in controlling which processes load code. Consider a file, DummyModule.jl, containing the following code:

```
module DummyModule

export MyType, f

mutable struct MyType
    a::Int
end

f(x) = x^2+1

println("loaded")

end
```

In order to refer to MyType across all processes, DummyModule.jl needs to be loaded on every process. Calling include("DummyModule.jl") loads it only on a single process. To load it on every process, use the @everywhere macro (starting Julia with julia -p 2):

```
julia> @everywhere include("DummyModule.jl")
loaded
      From worker 3:    loaded
      From worker 2:    loaded
```

As usual, this does not bring DummyModule into scope on any of the processes, which requires using or import. Moreover, when DummyModule is brought into scope on one process, it is not on any other:

```
julia> using .DummyModule

julia> MyType(7)
MyType(7)

julia> fetch(@spawnat 2 MyType(7))
ERROR: On worker 2:
UndefVarError: `MyType` not defined in `Main`
□

julia> fetch(@spawnat 2 DummyModule.MyType(7))
MyType(7)
```

However, it's still possible, for instance, to send a MyType to a process which has loaded DummyModule even if it's not in scope:

```
julia> put!(RemoteChannel(2), MyType(7))
RemoteChannel{Channel{Any}}(2, 1, 13)
```

A file can also be preloaded on multiple processes at startup with the `-L` flag, and a driver script can be used to drive the computation:

```
julia -p <n> -L file1.jl -L file2.jl driver.jl
```

The Julia process running the driver script in the example above has an `id` equal to 1, just like a process providing an interactive prompt.

Finally, if `DummyModule.jl` is not a standalone file but a package, then `using DummyModule` will *load* `DummyModule.jl` on all processes, but only bring it into scope on the process where `using` was called.

26.2 Starting and managing worker processes

The base Julia installation has in-built support for two types of clusters:

- A local cluster specified with the `-p` option as shown above.
- A cluster spanning machines using the `--machine-file` option. This uses a passwordless `ssh` login to start Julia worker processes (from the same path as the current host) on the specified machines. Each machine definition takes the form `[count*][user@]host[:port] [bind_addr[:port]]`. `user` defaults to current user, `port` to the standard ssh port. `count` is the number of workers to spawn on the node, and defaults to 1. The optional `bind-to bind_addr[:port]` specifies the IP address and port that other workers should use to connect to this worker.

Note

While Julia generally strives for backward compatibility, distribution of code to worker processes relies on `Serialization.serialize`. As pointed out in the corresponding documentation, this can not be guaranteed to work across different Julia versions, so it is advised that all workers on all machines use the same version.

Functions `addprocs`, `rmprocs`, `workers`, and others are available as a programmatic means of adding, removing and querying the processes in a cluster.

```
julia> using Distributed

julia> addprocs(2)
2-element Vector{Int64}:
 2
 3
```

Module `Distributed` must be explicitly loaded on the master process before invoking `addprocs`. It is automatically made available on the worker processes.

Note

Note that workers do not run a `~/.julia/config/startup.jl` startup script, nor do they synchronize their global state (such as command-line switches, global variables, new method definitions, and loaded modules) with any of the other running processes. You may use `addprocs(exeflags="--project")` to initialize a worker with a particular environment, and then `@everywhere using <modulename>` or `@everywhere include("file.jl")`.

Other types of clusters can be supported by writing your own custom `ClusterManager`, as described below in the ClusterManagers section.

26.3 Data Movement

Sending messages and moving data constitute most of the overhead in a distributed program. Reducing the number of messages and the amount of data sent is critical to achieving performance and scalability. To this end, it is important to understand the data movement performed by Julia's various distributed programming constructs.

`fetch` can be considered an explicit data movement operation, since it directly asks that an object be moved to the local machine. `@spawnat` (and a few related constructs) also moves data, but this is not as obvious, hence it can be called an implicit data movement operation. Consider these two approaches to constructing and squaring a random matrix:

Method 1:

```
julia> A = rand(1000,1000);

julia> Bref = @spawnat :any A^2;

[...]

julia> fetch(Bref);
```

Method 2:

```
julia> Bref = @spawnat :any rand(1000,1000)^2;

[...]

julia> fetch(Bref);
```

The difference seems trivial, but in fact is quite significant due to the behavior of `@spawnat`. In the first method, a random matrix is constructed locally, then sent to another process where it is squared. In the second method, a random matrix is both constructed and squared on another process. Therefore the second method sends much less data than the first.

In this toy example, the two methods are easy to distinguish and choose from. However, in a real program designing data movement might require more thought and likely some measurement. For example, if the first process needs matrix `A` then the first method might be better. Or, if computing `A` is expensive and only the current process has it, then moving it to another process might be unavoidable. Or, if the current process has very little to do between the `@spawnat` and `fetch(Bref)`, it might be better to eliminate the parallelism altogether. Or imagine `rand(1000,1000)` is replaced with a more expensive operation. Then it might make sense to add another `@spawnat` statement just for this step.

26.4 Global variables

Expressions executed remotely via `@spawnat`, or closures specified for remote execution using `remotecall` may refer to global variables. Global bindings under module `Main` are treated a little differently compared to global bindings in other modules. Consider the following code snippet:

```
A = rand(10,10)
remotecall_fetch(()->sum(A), 2)
```

In this case `sum` MUST be defined in the remote process. Note that `A` is a global variable defined in the local workspace. Worker 2 does not have a variable called `A` under `Main`. The act of shipping the closure `()->sum(A)` to worker 2 results in `Main.A` being defined on 2. `Main.A` continues to exist on worker 2 even after the call `remotecall_fetch` returns. Remote calls with embedded global references (under `Main` module only) manage globals as follows:

- New global bindings are created on destination workers if they are referenced as part of a remote call.
- Global constants are declared as constants on remote nodes too.
- Globals are re-sent to a destination worker only in the context of a remote call, and then only if its value has changed. Also, the cluster does not synchronize global bindings across nodes. For example:

  ```
  A = rand(10,10)
  remotecall_fetch(()->sum(A), 2) # worker 2
  A = rand(10,10)
  remotecall_fetch(()->sum(A), 3) # worker 3
  A = nothing
  ```

  Executing the above snippet results in `Main.A` on worker 2 having a different value from `Main.A` on worker 3, while the value of `Main.A` on node 1 is set to `nothing`.

As you may have realized, while memory associated with globals may be collected when they are reassigned on the master, no such action is taken on the workers as the bindings continue to be valid. `clear!` can be used to manually reassign specific globals on remote nodes to `nothing` once they are no longer required. This will release any memory associated with them as part of a regular garbage collection cycle.

Thus programs should be careful referencing globals in remote calls. In fact, it is preferable to avoid them altogether if possible. If you must reference globals, consider using `let` blocks to localize global variables.

For example:

```
julia> A = rand(10,10);

julia> remotecall_fetch(()->A, 2);

julia> B = rand(10,10);

julia> let B = B
           remotecall_fetch(()->B, 2)
       end;

julia> @fetchfrom 2 InteractiveUtils.varinfo()
name           size summary
———————— ————————— ————————————————————
A         800 bytes 10×10 Array{Float64,2}
Base                Module
Core                Module
Main                Module
```

As can be seen, global variable A is defined on worker 2, but B is captured as a local variable and hence a binding for B does not exist on worker 2.

26.5 Parallel Map and Loops

Fortunately, many useful parallel computations do not require data movement. A common example is a Monte Carlo simulation, where multiple processes can handle independent simulation trials simultaneously. We can use `@spawnat` to flip coins on two processes. First, write the following function in `count_heads.jl`:

```
function count_heads(n)
    c::Int = 0
    for i = 1:n
        c += rand(Bool)
    end
    c
end
```

The function `count_heads` simply adds together `n` random bits. Here is how we can perform some trials on two machines, and add together the results:

```
julia> @everywhere include_string(Main, $(read("count_heads.jl", String)), "count_heads.jl")

julia> a = @spawnat :any count_heads(100000000)
Future(2, 1, 6, nothing)

julia> b = @spawnat :any count_heads(100000000)
Future(3, 1, 7, nothing)

julia> fetch(a)+fetch(b)
100001564
```

This example demonstrates a powerful and often-used parallel programming pattern. Many iterations run independently over several processes, and then their results are combined using some function. The combination process is called a *reduction*, since it is generally tensor-rank-reducing: a vector of numbers is reduced to a single number, or a matrix is reduced to a single row or column, etc. In code, this typically looks like the pattern `x = f(x,v[i])`, where `x` is the accumulator, `f` is the reduction function, and the `v[i]` are the elements being reduced. It is desirable for `f` to be associative, so that it does not matter what order the operations are performed in.

Notice that our use of this pattern with `count_heads` can be generalized. We used two explicit `@spawnat` statements, which limits the parallelism to two processes. To run on any number of processes, we can use a *parallel for loop*, running in distributed memory, which can be written in Julia using `@distributed` like this:

```
nheads = @distributed (+) for i = 1:200000000
    Int(rand(Bool))
end
```

This construct implements the pattern of assigning iterations to multiple processes, and combining them with a specified reduction (in this case (+)). The result of each iteration is taken as the value of the last expression inside the loop. The whole parallel loop expression itself evaluates to the final answer.

Note that although parallel for loops look like serial for loops, their behavior is dramatically different. In particular, the iterations do not happen in a specified order, and writes to variables or arrays will not be globally visible since iterations run on different processes. Any variables used inside the parallel loop will be copied and broadcast to each process.

For example, the following code will not work as intended:

```
a = zeros(100000)
@distributed for i = 1:100000
    a[i] = i
end
```

This code will not initialize all of a, since each process will have a separate copy of it. Parallel for loops like these must be avoided. Fortunately, Shared Arrays can be used to get around this limitation:

```
using SharedArrays

a = SharedArray{Float64}(10)
@distributed for i = 1:10
    a[i] = i
end
```

Using "outside" variables in parallel loops is perfectly reasonable if the variables are read-only:

```
a = randn(1000)
@distributed (+) for i = 1:100000
    f(a[rand(1:end)])
end
```

Here each iteration applies f to a randomly-chosen sample from a vector a shared by all processes.

As you could see, the reduction operator can be omitted if it is not needed. In that case, the loop executes asynchronously, i.e. it spawns independent tasks on all available workers and returns an array of Future immediately without waiting for completion. The caller can wait for the Future completions at a later point by calling fetch on them, or wait for completion at the end of the loop by prefixing it with @sync, like @sync @distributed for.

In some cases no reduction operator is needed, and we merely wish to apply a function to all integers in some range (or, more generally, to all elements in some collection). This is another useful operation called *parallel map*, implemented in Julia as the pmap function. For example, we could compute the singular values of several large random matrices in parallel as follows:

```
julia> M = Matrix{Float64}[rand(1000,1000) for i = 1:10];

julia> pmap(svdvals, M);
```

Julia's `pmap` is designed for the case where each function call does a large amount of work. In contrast, `@distributed for` can handle situations where each iteration is tiny, perhaps merely summing two numbers. Only worker processes are used by both `pmap` and `@distributed for` for the parallel computation. In case of `@distributed for`, the final reduction is done on the calling process.

26.6 Remote References and AbstractChannels

Remote references always refer to an implementation of an `AbstractChannel`.

A concrete implementation of an `AbstractChannel` (like `Channel`), is required to implement `put!`, `take!`, `fetch`, `isready` and `wait`. The remote object referred to by a `Future` is stored in a `Channel{Any}(1)`, i.e., a `Channel` of size 1 capable of holding objects of `Any` type.

`RemoteChannel`, which is rewritable, can point to any type and size of channels, or any other implementation of an `AbstractChannel`.

The constructor `RemoteChannel(f::Function, pid)()` allows us to construct references to channels holding more than one value of a specific type. `f` is a function executed on `pid` and it must return an `AbstractChannel`.

For example, `RemoteChannel(()->Channel{Int}(10), pid)`, will return a reference to a channel of type `Int` and size 10. The channel exists on worker `pid`.

Methods `put!`, `take!`, `fetch`, `isready` and `wait` on a `RemoteChannel` are proxied onto the backing store on the remote process.

`RemoteChannel` can thus be used to refer to user implemented `AbstractChannel` objects. A simple example of this is the following `DictChannel` which uses a dictionary as its remote store:

```
julia> struct DictChannel{T} <: AbstractChannel{T}
           d::Dict
           cond_take::Threads.Condition    # waiting for data to become available
           DictChannel{T}() where {T} = new(Dict(), Threads.Condition())
           DictChannel() = DictChannel{Any}()
       end

julia> begin
       function Base.put!(D::DictChannel, k, v)
           @lock D.cond_take begin
               D.d[k] = v
               notify(D.cond_take)
           end
           return D
       end
       function Base.take!(D::DictChannel, k)
           @lock D.cond_take begin
               v = fetch(D, k)
               delete!(D.d, k)
               return v
           end
       end
       Base.isready(D::DictChannel) = @lock D.cond_take !isempty(D.d)
       Base.isready(D::DictChannel, k) = @lock D.cond_take haskey(D.d, k)
       function Base.fetch(D::DictChannel, k)
           @lock D.cond_take begin
```

```
                wait(D, k)
                return D.d[k]
            end
        end
        function Base.wait(D::DictChannel, k)
            @lock D.cond_take begin
                while !isready(D, k)
                    wait(D.cond_take)
                end
            end
        end
        end;

julia> d = DictChannel();

julia> isready(d)
false

julia> put!(d, :k, :v);

julia> isready(d, :k)
true

julia> fetch(d, :k)
:v

julia> wait(d, :k)

julia> take!(d, :k)
:v

julia> isready(d, :k)
false
```

26.7 Channels and RemoteChannels

- A `Channel` is local to a process. Worker 2 cannot directly refer to a `Channel` on worker 3 and vice-versa. A `RemoteChannel`, however, can put and take values across workers.
- A `RemoteChannel` can be thought of as a *handle* to a `Channel`.
- The process id, `pid`, associated with a `RemoteChannel` identifies the process where the backing store, i.e., the backing `Channel` exists.
- Any process with a reference to a `RemoteChannel` can put and take items from the channel. Data is automatically sent to (or retrieved from) the process a `RemoteChannel` is associated with.
- Serializing a `Channel` also serializes any data present in the channel. Deserializing it therefore effectively makes a copy of the original object.
- On the other hand, serializing a `RemoteChannel` only involves the serialization of an identifier that identifies the location and instance of `Channel` referred to by the handle. A deserialized `RemoteChannel` object (on any worker), therefore also points to the same backing store as the original.

The channels example from above can be modified for interprocess communication, as shown below.

We start 4 workers to process a single `jobs` remote channel. Jobs, identified by an id (`job_id`), are written to the channel. Each remotely executing task in this simulation reads a `job_id`, waits for a random amount of time and writes back a tuple of `job_id`, time taken and its own `pid` to the results channel. Finally all the `results` are printed out on the master process.

```
julia> addprocs(4); # add worker processes

julia> const jobs = RemoteChannel(()->Channel{Int}(32));

julia> const results = RemoteChannel(()->Channel{Tuple}(32));

julia> @everywhere function do_work(jobs, results) # define work function everywhere
           while true
               job_id = take!(jobs)
               exec_time = rand()
               sleep(exec_time) # simulates elapsed time doing actual work
               put!(results, (job_id, exec_time, myid()))
           end
       end

julia> function make_jobs(n)
           for i in 1:n
               put!(jobs, i)
           end
       end;

julia> n = 12;

julia> errormonitor(Threads.@spawn make_jobs(n)); # feed the jobs channel with "n" jobs

julia> for p in workers() # start tasks on the workers to process requests in parallel
           remote_do(do_work, p, jobs, results)
       end

julia> @elapsed while n > 0 # print out results
           job_id, exec_time, worker_id = take!(results)
           println("$job_id finished in $(round(exec_time; digits=2)) seconds on worker
           ↪  $worker_id")
           global n = n - 1
       end
1 finished in 0.18 seconds on worker 4
2 finished in 0.26 seconds on worker 5
6 finished in 0.12 seconds on worker 4
7 finished in 0.18 seconds on worker 4
5 finished in 0.35 seconds on worker 5
4 finished in 0.68 seconds on worker 2
3 finished in 0.73 seconds on worker 3
11 finished in 0.01 seconds on worker 3
12 finished in 0.02 seconds on worker 3
9 finished in 0.26 seconds on worker 5
8 finished in 0.57 seconds on worker 4
```

```
10 finished in 0.58 seconds on worker 2
0.055971741
```

Remote References and Distributed Garbage Collection

Objects referred to by remote references can be freed only when *all* held references in the cluster are deleted.

The node where the value is stored keeps track of which of the workers have a reference to it. Every time a `RemoteChannel` or a (unfetched) `Future` is serialized to a worker, the node pointed to by the reference is notified. And every time a `RemoteChannel` or a (unfetched) `Future` is garbage collected locally, the node owning the value is again notified. This is implemented in an internal cluster aware serializer. Remote references are only valid in the context of a running cluster. Serializing and deserializing references to and from regular `IO` objects is not supported.

The notifications are done via sending of "tracking" messages–an "add reference" message when a reference is serialized to a different process and a "delete reference" message when a reference is locally garbage collected.

Since `Future`s are write-once and cached locally, the act of `fetch`ing a `Future` also updates reference tracking information on the node owning the value.

The node which owns the value frees it once all references to it are cleared.

With `Future`s, serializing an already fetched `Future` to a different node also sends the value since the original remote store may have collected the value by this time.

It is important to note that *when* an object is locally garbage collected depends on the size of the object and the current memory pressure in the system.

In case of remote references, the size of the local reference object is quite small, while the value stored on the remote node may be quite large. Since the local object may not be collected immediately, it is a good practice to explicitly call `finalize` on local instances of a `RemoteChannel`, or on unfetched `Future`s. Since calling `fetch` on a `Future` also removes its reference from the remote store, this is not required on fetched `Future`s. Explicitly calling `finalize` results in an immediate message sent to the remote node to go ahead and remove its reference to the value.

Once finalized, a reference becomes invalid and cannot be used in any further calls.

26.8 Local invocations

Data is necessarily copied over to the remote node for execution. This is the case for both remotecalls and when data is stored to a `RemoteChannel` / `Future` on a different node. As expected, this results in a copy of the serialized objects on the remote node. However, when the destination node is the local node, i.e. the calling process id is the same as the remote node id, it is executed as a local call. It is usually (not always) executed in a different task - but there is no serialization/deserialization of data. Consequently, the call refers to the same object instances as passed - no copies are created. This behavior is highlighted below:

```
julia> using Distributed

julia> rc = RemoteChannel(()->Channel(3));   # RemoteChannel created on local node

julia> v = [0];
```

```
julia> for i in 1:3
           v[1] = i                          # Reusing `v`
           put!(rc, v)
       end;

julia> result = [take!(rc) for _ in 1:3];

julia> println(result);
[[3], [3], [3]]

julia> println("Num Unique objects : ", length(unique(map(objectid, result))));
Num Unique objects : 1

julia> addprocs(1);

julia> rc = RemoteChannel(()->Channel(3), workers()[1]);   # RemoteChannel created on remote node

julia> v = [0];

julia> for i in 1:3
           v[1] = i
           put!(rc, v)
       end;

julia> result = [take!(rc) for _ in 1:3];

julia> println(result);
[[1], [2], [3]]

julia> println("Num Unique objects : ", length(unique(map(objectid, result))));
Num Unique objects : 3
```

As can be seen, `put!` on a locally owned `RemoteChannel` with the same object v modified between calls results in the same single object instance stored. As opposed to copies of v being created when the node owning `rc` is a different node.

It is to be noted that this is generally not an issue. It is something to be factored in only if the object is both being stored locally and modified post the call. In such cases it may be appropriate to store a deepcopy of the object.

This is also true for remotecalls on the local node as seen in the following example:

```
julia> using Distributed; addprocs(1);

julia> v = [0];

julia> v2 = remotecall_fetch(x->(x[1] = 1; x), myid(), v);     # Executed on local node

julia> println("v=$v, v2=$v2, ", v === v2);
v=[1], v2=[1], true

julia> v = [0];
```

```
julia> v2 = remotecall_fetch(x->(x[1] = 1; x), workers()[1], v); # Executed on remote node

julia> println("v=$v, v2=$v2, ", v === v2);
v=[0], v2=[1], false
```

As can be seen once again, a remote call onto the local node behaves just like a direct invocation. The call modifies local objects passed as arguments. In the remote invocation, it operates on a copy of the arguments.

To repeat, in general this is not an issue. If the local node is also being used as a compute node, and the arguments used post the call, this behavior needs to be factored in and if required deep copies of arguments must be passed to the call invoked on the local node. Calls on remote nodes will always operate on copies of arguments.

26.9 Shared Arrays

Shared Arrays use system shared memory to map the same array across many processes. A `SharedArray` is a good choice when you want to have a large amount of data jointly accessible to two or more processes on the same machine. Shared Array support is available via the module `SharedArrays`, which must be explicitly loaded on all participating workers.

A complementary data structure is provided by the external package `DistributedArrays.jl` in the form of a `DArray`. While there are some similarities to a `SharedArray`, the behavior of a `DArray` is quite different. In a `SharedArray`, each "participating" process has access to the entire array; in contrast, in a `DArray`, each process has local access to just a chunk of the data, and no two processes share the same chunk.

`SharedArray` indexing (assignment and accessing values) works just as with regular arrays, and is efficient because the underlying memory is available to the local process. Therefore, most algorithms work naturally on `SharedArray`s, albeit in single-process mode. In cases where an algorithm insists on an `Array` input, the underlying array can be retrieved from a `SharedArray` by calling `sdata`. For other `AbstractArray` types, `sdata` just returns the object itself, so it's safe to use `sdata` on any `Array`-type object.

The constructor for a shared array is of the form:

```
SharedArray{T,N}(dims::NTuple; init=false, pids=Int[])
```

which creates an `N`-dimensional shared array of a bits type `T` and size `dims` across the processes specified by `pids`. Unlike distributed arrays, a shared array is accessible only from those participating workers specified by the `pids` named argument (and the creating process too, if it is on the same host). Note that only elements that are `isbits` are supported in a SharedArray.

If an `init` function, of signature `initfn(S::SharedArray)`, is specified, it is called on all the participating workers. You can specify that each worker runs the `init` function on a distinct portion of the array, thereby parallelizing initialization.

Here's a brief example:

```
julia> using Distributed

julia> addprocs(3)
```

```
3-element Vector{Int64}:
 2
 3
 4

julia> @everywhere using SharedArrays

julia> S = SharedArray{Int,2}((3,4), init = S -> S[localindices(S)] = repeat([myid()],
↪  length(localindices(S))))
3×4 SharedMatrix{Int64}:
 2  2  3  4
 2  3  3  4
 2  3  4  4

julia> S[3,2] = 7
7

julia> S
3×4 SharedMatrix{Int64}:
 2  2  3  4
 2  3  3  4
 2  7  4  4
```

`SharedArrays.localindices` provides disjoint one-dimensional ranges of indices, and is sometimes convenient for splitting up tasks among processes. You can, of course, divide the work any way you wish:

```
julia> S = SharedArray{Int,2}((3,4), init = S -> S[indexpids(S):length(procs(S)):length(S)] =
↪  repeat([myid()], length( indexpids(S):length(procs(S)):length(S))))
3×4 SharedMatrix{Int64}:
 2  2  2  2
 3  3  3  3
 4  4  4  4
```

Since all processes have access to the underlying data, you do have to be careful not to set up conflicts. For example:

```
@sync begin
    for p in procs(S)
        Threads.@spawn begin
            remotecall_wait(fill!, p, S, p)
        end
    end
end
```

would result in undefined behavior. Because each process fills the *entire* array with its own `pid`, whichever process is the last to execute (for any particular element of S) will have its `pid` retained.

As a more extended and complex example, consider running the following "kernel" in parallel:

```
q[i,j,t+1] = q[i,j,t] + u[i,j,t]
```

In this case, if we try to split up the work using a one-dimensional index, we are likely to run into trouble: if `q[i,j,t]` is near the end of the block assigned to one worker and `q[i,j,t+1]` is near the beginning of the block assigned to another, it's very likely that `q[i,j,t]` will not be ready at the time it's needed for computing `q[i,j,t+1]`. In such cases, one is better off chunking the array manually. Let's split along the second dimension. Define a function that returns the `(irange, jrange)` indices assigned to this worker:

```
julia> @everywhere function myrange(q::SharedArray)
           idx = indexpids(q)
           if idx == 0 # This worker is not assigned a piece
               return 1:0, 1:0
           end
           nchunks = length(procs(q))
           splits = [round(Int, s) for s in range(0, stop=size(q,2), length=nchunks+1)]
           1:size(q,1), splits[idx]+1:splits[idx+1]
       end
```

Next, define the kernel:

```
julia> @everywhere function advection_chunk!(q, u, irange, jrange, trange)
           @show (irange, jrange, trange)  # display so we can see what's happening
           for t in trange, j in jrange, i in irange
               q[i,j,t+1] = q[i,j,t] + u[i,j,t]
           end
           q
       end
```

We also define a convenience wrapper for a `SharedArray` implementation

```
julia> @everywhere advection_shared_chunk!(q, u) =
           advection_chunk!(q, u, myrange(q)..., 1:size(q,3)-1)
```

Now let's compare three different versions, one that runs in a single process:

```
julia> advection_serial!(q, u) = advection_chunk!(q, u, 1:size(q,1), 1:size(q,2), 1:size(q,3)-1);
```

one that uses `@distributed`:

```
julia> function advection_parallel!(q, u)
           for t = 1:size(q,3)-1
               @sync @distributed for j = 1:size(q,2)
                   for i = 1:size(q,1)
                       q[i,j,t+1]= q[i,j,t] + u[i,j,t]
                   end
               end
           end
           q
       end;
```

and one that delegates in chunks:

```
julia> function advection_shared!(q, u)
           @sync begin
               for p in procs(q)
                   Threads.@spawn remotecall_wait(advection_shared_chunk!, p, q, u)
               end
           end
           q
       end;
```

If we create SharedArrays and time these functions, we get the following results (with `julia -p 4`):

```
julia> q = SharedArray{Float64,3}((500,500,500));

julia> u = SharedArray{Float64,3}((500,500,500));
```

Run the functions once to JIT-compile and `@time` them on the second run:

```
julia> @time advection_serial!(q, u);
(irange,jrange,trange) = (1:500,1:500,1:499)
 830.220 milliseconds (216 allocations: 13820 bytes)

julia> @time advection_parallel!(q, u);
   2.495 seconds      (3999 k allocations: 289 MB, 2.09% gc time)

julia> @time advection_shared!(q,u);
        From worker 2:       (irange,jrange,trange) = (1:500,1:125,1:499)
        From worker 4:       (irange,jrange,trange) = (1:500,251:375,1:499)
        From worker 3:       (irange,jrange,trange) = (1:500,126:250,1:499)
        From worker 5:       (irange,jrange,trange) = (1:500,376:500,1:499)
 238.119 milliseconds (2264 allocations: 169 KB)
```

The biggest advantage of `advection_shared!` is that it minimizes traffic among the workers, allowing each to compute for an extended time on the assigned piece.

Shared Arrays and Distributed Garbage Collection

Like remote references, shared arrays are also dependent on garbage collection on the creating node to release references from all participating workers. Code which creates many short lived shared array objects would benefit from explicitly finalizing these objects as soon as possible. This results in both memory and file handles mapping the shared segment being released sooner.

26.10 ClusterManagers

The launching, management and networking of Julia processes into a logical cluster is done via cluster managers. A `ClusterManager` is responsible for

- launching worker processes in a cluster environment
- managing events during the lifetime of each worker

- optionally, providing data transport

A Julia cluster has the following characteristics:

- The initial Julia process, also called the `master`, is special and has an `id` of `1`.
- Only the `master` process can add or remove worker processes.
- All processes can directly communicate with each other.

Connections between workers (using the in-built TCP/IP transport) is established in the following manner:

- `addprocs` is called on the master process with a `ClusterManager` object.
- `addprocs` calls the appropriate `launch` method which spawns required number of worker processes on appropriate machines.
- Each worker starts listening on a free port and writes out its host and port information to `stdout`.
- The cluster manager captures the `stdout` of each worker and makes it available to the master process.
- The master process parses this information and sets up TCP/IP connections to each worker.
- Every worker is also notified of other workers in the cluster.
- Each worker connects to all workers whose `id` is less than the worker's own `id`.
- In this way a mesh network is established, wherein every worker is directly connected with every other worker.

While the default transport layer uses plain `TCPSocket`, it is possible for a Julia cluster to provide its own transport.

Julia provides two in-built cluster managers:

- `LocalManager`, used when `addprocs()` or `addprocs(np::Integer)` are called
- `SSHManager`, used when `addprocs(hostnames::Array)` is called with a list of hostnames

`LocalManager` is used to launch additional workers on the same host, thereby leveraging multi-core and multi-processor hardware.

Thus, a minimal cluster manager would need to:

- be a subtype of the abstract `ClusterManager`
- implement `launch`, a method responsible for launching new workers
- implement `manage`, which is called at various events during a worker's lifetime (for example, sending an interrupt signal)

`addprocs(manager::FooManager)` requires `FooManager` to implement:

```
function launch(manager::FooManager, params::Dict, launched::Array, c::Condition)
    [...]
end

function manage(manager::FooManager, id::Integer, config::WorkerConfig, op::Symbol)
    [...]
end
```

As an example let us see how the `LocalManager`, the manager responsible for starting workers on the same host, is implemented:

```
struct LocalManager <: ClusterManager
    np::Integer
end

function launch(manager::LocalManager, params::Dict, launched::Array, c::Condition)
    [...]
end

function manage(manager::LocalManager, id::Integer, config::WorkerConfig, op::Symbol)
    [...]
end
```

The `launch` method takes the following arguments:

- `manager::ClusterManager`: the cluster manager that `addprocs` is called with
- `params::Dict`: all the keyword arguments passed to `addprocs`
- `launched::Array`: the array to append one or more `WorkerConfig` objects to
- `c::Condition`: the condition variable to be notified as and when workers are launched

The `launch` method is called asynchronously in a separate task. The termination of this task signals that all requested workers have been launched. Hence the `launch` function MUST exit as soon as all the requested workers have been launched.

Newly launched workers are connected to each other and the master process in an all-to-all manner. Specifying the command line argument `--worker[=<cookie>]` results in the launched processes initializing themselves as workers and connections being set up via TCP/IP sockets.

All workers in a cluster share the same cookie as the master. When the cookie is unspecified, i.e, with the `--worker` option, the worker tries to read it from its standard input. `LocalManager` and `SSHManager` both pass the cookie to newly launched workers via their standard inputs.

By default a worker will listen on a free port at the address returned by a call to `getipaddr()`. A specific address to listen on may be specified by optional argument `--bind-to bind_addr[:port]`. This is useful for multi-homed hosts.

As an example of a non-TCP/IP transport, an implementation may choose to use MPI, in which case `--worker` must NOT be specified. Instead, newly launched workers should call `init_worker(cookie)` before using any of the parallel constructs.

For every worker launched, the `launch` method must add a `WorkerConfig` object (with appropriate fields initialized) to `launched`

```
mutable struct WorkerConfig
    # Common fields relevant to all cluster managers
    io::Union{IO, Nothing}
    host::Union{AbstractString, Nothing}
    port::Union{Integer, Nothing}

    # Used when launching additional workers at a host
    count::Union{Int, Symbol, Nothing}
    exename::Union{AbstractString, Cmd, Nothing}
    exeflags::Union{Cmd, Nothing}

    # External cluster managers can use this to store information at a per-worker level
    # Can be a dict if multiple fields need to be stored.
    userdata::Any

    # SSHManager / SSH tunnel connections to workers
    tunnel::Union{Bool, Nothing}
    bind_addr::Union{AbstractString, Nothing}
    sshflags::Union{Cmd, Nothing}
    max_parallel::Union{Integer, Nothing}

    # Used by Local/SSH managers
    connect_at::Any

    [...]
end
```

Most of the fields in `WorkerConfig` are used by the inbuilt managers. Custom cluster managers would typically specify only `io` or `host` / `port`:

- If `io` is specified, it is used to read host/port information. A Julia worker prints out its bind address and port at startup. This allows Julia workers to listen on any free port available instead of requiring worker ports to be configured manually.
- If `io` is not specified, `host` and `port` are used to connect.
- `count`, `exename` and `exeflags` are relevant for launching additional workers from a worker. For example, a cluster manager may launch a single worker per node, and use that to launch additional workers.
  - `count` with an integer value `n` will launch a total of `n` workers.
  - `count` with a value of `:auto` will launch as many workers as the number of CPU threads (logical cores) on that machine.
  - `exename` is the name of the `julia` executable including the full path.

  - exeflags should be set to the required command line arguments for new workers.

- tunnel, bind_addr, sshflags and max_parallel are used when a ssh tunnel is required to connect to the workers from the master process.
- userdata is provided for custom cluster managers to store their own worker-specific information.

manage(manager::FooManager, id::Integer, config::WorkerConfig, op::Symbol) is called at different times during the worker's lifetime with appropriate op values:

- with :register/:deregister when a worker is added / removed from the Julia worker pool.
- with :interrupt when interrupt(workers) is called. The ClusterManager should signal the appropriate worker with an interrupt signal.
- with :finalize for cleanup purposes.

Cluster Managers with Custom Transports

Replacing the default TCP/IP all-to-all socket connections with a custom transport layer is a little more involved. Each Julia process has as many communication tasks as the workers it is connected to. For example, consider a Julia cluster of 32 processes in an all-to-all mesh network:

- Each Julia process thus has 31 communication tasks.
- Each task handles all incoming messages from a single remote worker in a message-processing loop.
- The message-processing loop waits on an IO object (for example, a TCPSocket in the default implementation), reads an entire message, processes it and waits for the next one.
- Sending messages to a process is done directly from any Julia task–not just communication tasks–again, via the appropriate IO object.

Replacing the default transport requires the new implementation to set up connections to remote workers and to provide appropriate IO objects that the message-processing loops can wait on. The manager-specific callbacks to be implemented are:

```
connect(manager::FooManager, pid::Integer, config::WorkerConfig)
kill(manager::FooManager, pid::Int, config::WorkerConfig)
```

The default implementation (which uses TCP/IP sockets) is implemented as connect(manager::ClusterManager, pid::Integer, config::WorkerConfig).

connect should return a pair of IO objects, one for reading data sent from worker pid, and the other to write data that needs to be sent to worker pid. Custom cluster managers can use an in-memory BufferStream as the plumbing to proxy data between the custom, possibly non-IO transport and Julia's in-built parallel infrastructure.

A BufferStream is an in-memory IOBuffer which behaves like an IO–it is a stream which can be handled asynchronously.

The folder `clustermanager/0mq` in the Examples repository contains an example of using ZeroMQ to connect Julia workers in a star topology with a 0MQ broker in the middle. Note: The Julia processes are still all *logically* connected to each other–any worker can message any other worker directly without any awareness of 0MQ being used as the transport layer.

When using custom transports:

- Julia workers must NOT be started with `--worker`. Starting with `--worker` will result in the newly launched workers defaulting to the TCP/IP socket transport implementation.
- For every incoming logical connection with a worker, `Base.process_messages(rd::IO, wr::IO)()` must be called. This launches a new task that handles reading and writing of messages from/to the worker represented by the `IO` objects.
- `init_worker(cookie, manager::FooManager)` *must* be called as part of worker process initialization.
- Field `connect_at::Any` in `WorkerConfig` can be set by the cluster manager when `launch` is called. The value of this field is passed in all `connect` callbacks. Typically, it carries information on *how to connect* to a worker. For example, the TCP/IP socket transport uses this field to specify the `(host, port)` tuple at which to connect to a worker.

`kill(manager, pid, config)` is called to remove a worker from the cluster. On the master process, the corresponding `IO` objects must be closed by the implementation to ensure proper cleanup. The default implementation simply executes an `exit()` call on the specified remote worker.

The Examples folder `clustermanager/simple` is an example that shows a simple implementation using UNIX domain sockets for cluster setup.

Network Requirements for LocalManager and SSHManager

Julia clusters are designed to be executed on already secured environments on infrastructure such as local laptops, departmental clusters, or even the cloud. This section covers network security requirements for the inbuilt `LocalManager` and `SSHManager`:

- The master process does not listen on any port. It only connects out to the workers.
- Each worker binds to only one of the local interfaces and listens on an ephemeral port number assigned by the OS.
- `LocalManager`, used by `addprocs(N)`, by default binds only to the loopback interface. This means that workers started later on remote hosts (or by anyone with malicious intentions) are unable to connect to the cluster. An `addprocs(4)` followed by an `addprocs(["remote_host"])` will fail. Some users may need to create a cluster comprising their local system and a few remote systems. This can be done by explicitly requesting `LocalManager` to bind to an external network interface via the `restrict` keyword argument: `addprocs(4; restrict=false)`.
- `SSHManager`, used by `addprocs(list_of_remote_hosts)`, launches workers on remote hosts via SSH. By default SSH is only used to launch Julia workers. Subsequent master-worker and worker-worker connections use plain, unencrypted TCP/IP sockets. The remote hosts must have passwordless login enabled. Additional SSH flags or credentials may be specified via keyword argument `sshflags`.

- `addprocs(list_of_remote_hosts; tunnel=true, sshflags=<ssh keys and other flags>)` is useful when we wish to use SSH connections for master-worker too. A typical scenario for this is a local laptop running the Julia REPL (i.e., the master) with the rest of the cluster on the cloud, say on Amazon EC2. In this case only port 22 needs to be opened at the remote cluster coupled with SSH client authenticated via public key infrastructure (PKI). Authentication credentials can be supplied via `sshflags`, for example `sshflags=`-i <keyfile>`.

  In an all-to-all topology (the default), all workers connect to each other via plain TCP sockets. The security policy on the cluster nodes must thus ensure free connectivity between workers for the ephemeral port range (varies by OS).

  Securing and encrypting all worker-worker traffic (via SSH) or encrypting individual messages can be done via a custom `ClusterManager`.

- If you specify `multiplex=true` as an option to `addprocs`, SSH multiplexing is used to create a tunnel between the master and workers. If you have configured SSH multiplexing on your own and the connection has already been established, SSH multiplexing is used regardless of `multiplex` option. If multiplexing is enabled, forwarding is set by using the existing connection (`-O forward` option in ssh). This is beneficial if your servers require password authentication; you can avoid authentication in Julia by logging in to the server ahead of `addprocs`. The control socket will be located at `~/.ssh/julia-%r@%h:%p` during the session unless the existing multiplexing connection is used. Note that bandwidth may be limited if you create multiple processes on a node and enable multiplexing, because in that case processes share a single multiplexing TCP connection.

Cluster Cookie

All processes in a cluster share the same cookie which, by default, is a randomly generated string on the master process:

- `cluster_cookie()` returns the cookie, while `cluster_cookie(cookie)()` sets it and returns the new cookie.
- All connections are authenticated on both sides to ensure that only workers started by the master are allowed to connect to each other.
- The cookie may be passed to the workers at startup via argument `--worker=<cookie>`. If argument `--worker` is specified without the cookie, the worker tries to read the cookie from its standard input (`stdin`). The `stdin` is closed immediately after the cookie is retrieved.
- `ClusterManagers` can retrieve the cookie on the master by calling `cluster_cookie()`. Cluster managers not using the default TCP/IP transport (and hence not specifying `--worker`) must call `init_worker(cookie, manager)` with the same cookie as on the master.

Note that environments requiring higher levels of security can implement this via a custom `ClusterManager`. For example, cookies can be pre-shared and hence not specified as a startup argument.

26.11 Specifying Network Topology (Experimental)

The keyword argument `topology` passed to `addprocs` is used to specify how the workers must be connected to each other:

- `:all_to_all`, the default: all workers are connected to each other.
- `:master_worker`: only the driver process, i.e. `pid 1`, has connections to the workers.
- `:custom`: the `launch` method of the cluster manager specifies the connection topology via the fields `ident` and `connect_idents` in `WorkerConfig`. A worker with a cluster-manager-provided identity `ident` will connect to all workers specified in `connect_idents`.

Keyword argument `lazy=true|false` only affects `topology` option `:all_to_all`. If `true`, the cluster starts off with the master connected to all workers. Specific worker-worker connections are established at the first remote invocation between two workers. This helps in reducing initial resources allocated for intra-cluster communication. Connections are setup depending on the runtime requirements of a parallel program. Default value for `lazy` is `true`.

Currently, sending a message between unconnected workers results in an error. This behaviour, as with the functionality and interface, should be considered experimental in nature and may change in future releases.

26.12 Noteworthy external packages

Outside of Julia parallelism there are plenty of external packages that should be mentioned. For example, `MPI.jl` is a Julia wrapper for the `MPI` protocol, `Dagger.jl` provides functionality similar to Python's Dask, and `DistributedArrays.jl` provides array operations distributed across workers, as outlined above.

A mention must be made of Julia's GPU programming ecosystem, which includes:

1. CUDA.jl wraps the various CUDA libraries and supports compiling Julia kernels for Nvidia GPUs.
2. oneAPI.jl wraps the oneAPI unified programming model, and supports executing Julia kernels on supported accelerators. Currently only Linux is supported.
3. AMDGPU.jl wraps the AMD ROCm libraries and supports compiling Julia kernels for AMD GPUs. Currently only Linux is supported.
4. High-level libraries like KernelAbstractions.jl, Tullio.jl and ArrayFire.jl.

In the following example we will use both `DistributedArrays.jl` and `CUDA.jl` to distribute an array across multiple processes by first casting it through `distribute()` and `CuArray()`.

Remember when importing `DistributedArrays.jl` to import it across all processes using `@everywhere`

```
$ ./julia -p 4

julia> addprocs()

julia> @everywhere using DistributedArrays

julia> using CUDA

julia> B = ones(10_000) ./ 2;

julia> A = ones(10_000) .* π;
```

```
julia> C = 2 .* A ./ B;

julia> all(C .≈ 4*π)
true

julia> typeof(C)
Vector{Float64} (alias for Array{Float64, 1})

julia> dB = distribute(B);

julia> dA = distribute(A);

julia> dC = 2 .* dA ./ dB;

julia> all(dC .≈ 4*π)
true

julia> typeof(dC)
DistributedArrays.DArray{Float64,1,Vector{Float64}}

julia> cuB = CuArray(B);

julia> cuA = CuArray(A);

julia> cuC = 2 .* cuA ./ cuB;

julia> all(cuC .≈ 4*π);
true

julia> typeof(cuC)
CuArray{Float64,1}
```

In the following example we will use both `DistributedArrays.jl` and `CUDA.jl` to distribute an array across multiple processes and call a generic function on it.

```
function power_method(M, v)
    for i in 1:100
        v = M*v
        v /= norm(v)
    end

    return v, norm(M*v) / norm(v)  # or  (M*v) ./ v
end
```

`power_method` repeatedly creates a new vector and normalizes it. We have not specified any type signature in function declaration, let's see if it works with the aforementioned datatypes:

```
julia> M = [2. 1; 1 1];

julia> v = rand(2)
```

```
2-element Vector{Float64}:
0.40395
0.445877

julia> power_method(M,v)
([0.850651, 0.525731], 2.618033988749895)

julia> cuM = CuArray(M);

julia> cuv = CuArray(v);

julia> curesult = power_method(cuM, cuv);

julia> typeof(curesult)
CuArray{Float64,1}

julia> dM = distribute(M);

julia> dv = distribute(v);

julia> dC = power_method(dM, dv);

julia> typeof(dC)
Tuple{DistributedArrays.DArray{Float64,1,Vector{Float64}},Float64}
```

To end this short exposure to external packages, we can consider `MPI.jl`, a Julia wrapper of the MPI protocol. As it would take too long to consider every inner function, it would be better to simply appreciate the approach used to implement the protocol.

Consider this toy script which simply calls each subprocess, instantiate its rank and when the master process is reached, performs the ranks' sum

```
import MPI

MPI.Init()

comm = MPI.COMM_WORLD
MPI.Barrier(comm)

root = 0
r = MPI.Comm_rank(comm)

sr = MPI.Reduce(r, MPI.SUM, root, comm)

if(MPI.Comm_rank(comm) == root)
   @printf("sum of ranks: %s\n", sr)
end

MPI.Finalize()
```

```
mpirun -np 4 ./julia example.jl
```

[1] In this context, MPI refers to the MPI-1 standard. Beginning with MPI-2, the MPI standards committee introduced a new set

of communication mechanisms, collectively referred to as Remote Memory Access (RMA). The motivation for adding rma to the MPI standard was to facilitate one-sided communication patterns. For additional information on the latest MPI standard, see https://mpi-forum.org/docs.

Chapter 27

Running External Programs

Julia borrows backtick notation for commands from the shell, Perl, and Ruby. However, in Julia, writing

```
julia> `echo hello`
`echo hello`
```

differs in several aspects from the behavior in various shells, Perl, or Ruby:

- Instead of immediately running the command, backticks create a `Cmd` object to represent the command. You can use this object to connect the command to others via pipes, `run` it, and `read` or `write` to it.
- When the command is run, Julia does not capture its output unless you specifically arrange for it to. Instead, the output of the command by default goes to `stdout` as it would using `libc`'s `system` call.
- The command is never run with a shell. Instead, Julia parses the command syntax directly, appropriately interpolating variables and splitting on words as the shell would, respecting shell quoting syntax. The command is run as `julia`'s immediate child process, using `fork` and exec calls.

Note

The following assumes a Posix environment as on Linux or MacOS. On Windows, many similar commands, such as `echo` and `dir`, are not external programs and instead are built into the shell `cmd.exe` itself. One option to run these commands is to invoke `cmd.exe`, for example `cmd /C echo hello`. Alternatively Julia can be run inside a Posix environment such as Cygwin.

Here's a simple example of running an external program:

```
julia> mycommand = `echo hello`
`echo hello`

julia> typeof(mycommand)
Cmd

julia> run(mycommand);
hello
```

The `hello` is the output of the `echo` command, sent to `stdout`. If the external command fails to run successfully, the run method throws an `ProcessFailedException`.

If you want to read the output of the external command, `read` or `readchomp` can be used instead:

```
julia> read(`echo hello`, String)
"hello\n"

julia> readchomp(`echo hello`)
"hello"
```

More generally, you can use `open` to read from or write to an external command.

```
julia> open(`less`, "w", stdout) do io
           for i = 1:3
               println(io, i)
           end
       end
1
2
3
```

The program name and the individual arguments in a command can be accessed and iterated over as if the command were an array of strings:

```
julia> collect(`echo "foo bar"`)
2-element Vector{String}:
 "echo"
 "foo bar"

julia> `echo "foo bar"`[2]
"foo bar"
```

You can also pass a `IOBuffer`, and later read from it:

```
julia> io = PipeBuffer(); # PipeBuffer is a type of IOBuffer

julia> run(`echo world`, devnull, io, stderr);

julia> readlines(io)
1-element Vector{String}:
 "world"
```

27.1 Interpolation

Suppose you want to do something a bit more complicated and use the name of a file in the variable `file` as an argument to a command. You can use `$` for interpolation much as you would in a string literal (see Strings):

```
julia> file = "/etc/passwd"
"/etc/passwd"

julia> `sort $file`
`sort /etc/passwd`
```

A common pitfall when running external programs via a shell is that if a file name contains characters that are special to the shell, they may cause undesirable behavior. Suppose, for example, rather than `/etc/passwd`, we wanted to sort the contents of the file `/Volumes/External HD/data.csv`. Let's try it:

```
julia> file = "/Volumes/External HD/data.csv"
"/Volumes/External HD/data.csv"

julia> `sort $file`
`sort '/Volumes/External HD/data.csv'`
```

How did the file name get quoted? Julia knows that `file` is meant to be interpolated as a single argument, so it quotes the word for you. Actually, that is not quite accurate: the value of `file` is never interpreted by a shell, so there's no need for actual quoting; the quotes are inserted only for presentation to the user. This will even work if you interpolate a value as part of a shell word:

```
julia> path = "/Volumes/External HD"
"/Volumes/External HD"

julia> name = "data"
"data"

julia> ext = "csv"
"csv"

julia> `sort $path/$name.$ext`
`sort '/Volumes/External HD/data.csv'`
```

As you can see, the space in the `path` variable is appropriately escaped. But what if you *want* to interpolate multiple words? In that case, just use an array (or any other iterable container):

```
julia> files = ["/etc/passwd","/Volumes/External HD/data.csv"]
2-element Vector{String}:
 "/etc/passwd"
 "/Volumes/External HD/data.csv"

julia> `grep foo $files`
`grep foo /etc/passwd '/Volumes/External HD/data.csv'`
```

If you interpolate an array as part of a shell word, Julia emulates the shell's `{a,b,c}` argument generation:

```
julia> names = ["foo","bar","baz"]
3-element Vector{String}:
 "foo"
```

```
 "bar"
 "baz"

julia> `grep xylophone $names.txt`
`grep xylophone foo.txt bar.txt baz.txt`
```

Moreover, if you interpolate multiple arrays into the same word, the shell's Cartesian product generation behavior is emulated:

```
julia> names = ["foo","bar","baz"]
3-element Vector{String}:
 "foo"
 "bar"
 "baz"

julia> exts = ["aux","log"]
2-element Vector{String}:
 "aux"
 "log"

julia> `rm -f $names.$exts`
`rm -f foo.aux foo.log bar.aux bar.log baz.aux baz.log`
```

Since you can interpolate literal arrays, you can use this generative functionality without needing to create temporary array objects first:

```
julia> `rm -rf $["foo","bar","baz","qux"].$["aux","log","pdf"]`
`rm -rf foo.aux foo.log foo.pdf bar.aux bar.log bar.pdf baz.aux baz.log baz.pdf qux.aux qux.log
↪  qux.pdf`
```

27.2 Quoting

Inevitably, one wants to write commands that aren't quite so simple, and it becomes necessary to use quotes. Here's a simple example of a Perl one-liner at a shell prompt:

```
sh$ perl -le '$|=1; for (0..3) { print }'
0
1
2
3
```

The Perl expression needs to be in single quotes for two reasons: so that spaces don't break the expression into multiple shell words, and so that uses of Perl variables like `$|` (yes, that's the name of a variable in Perl), don't cause interpolation. In other instances, you may want to use double quotes so that interpolation *does* occur:

```
sh$ first="A"
sh$ second="B"
sh$ perl -le '$|=1; print for @ARGV' "1: $first" "2: $second"
1: A
2: B
```

In general, the Julia backtick syntax is carefully designed so that you can just cut-and-paste shell commands as is into backticks and they will work: the escaping, quoting, and interpolation behaviors are the same as the shell's. The only difference is that the interpolation is integrated and aware of Julia's notion of what is a single string value, and what is a container for multiple values. Let's try the above two examples in Julia:

```
julia> A = `perl -le '$|=1; for (0..3) { print }'`
`perl -le '$|=1; for (0..3) { print }'`

julia> run(A);
0
1
2
3

julia> first = "A"; second = "B";

julia> B = `perl -le 'print for @ARGV' "1: $first" "2: $second"`
`perl -le 'print for @ARGV' '1: A' '2: B'`

julia> run(B);
1: A
2: B
```

The results are identical, and Julia's interpolation behavior mimics the shell's with some improvements due to the fact that Julia supports first-class iterable objects while most shells use strings split on spaces for this, which introduces ambiguities. When trying to port shell commands to Julia, try cut and pasting first. Since Julia shows commands to you before running them, you can easily and safely just examine its interpretation without doing any damage.

27.3 Pipelines

Shell metacharacters, such as `|`, &, and >, need to be quoted (or escaped) inside of Julia's backticks:

```
julia> run(`echo hello '|' sort`);
hello | sort

julia> run(`echo hello \| sort`);
hello | sort
```

This expression invokes the `echo` command with three words as arguments: `hello`, `|`, and `sort`. The result is that a single line is printed: `hello | sort`. How, then, does one construct a pipeline? Instead of using `'|'` inside of backticks, one uses `pipeline`:

```
julia> run(pipeline(`echo hello`, `sort`));
hello
```

This pipes the output of the `echo` command to the `sort` command. Of course, this isn't terribly interesting since there's only one line to sort, but we can certainly do much more interesting things:

```
julia> run(pipeline(`cut -d: -f3 /etc/passwd`, `sort -n`, `tail -n5`))
210
211
212
213
214
```

This prints the highest five user IDs on a UNIX system. The `cut`, `sort` and `tail` commands are all spawned as immediate children of the current `julia` process, with no intervening shell process. Julia itself does the work to setup pipes and connect file descriptors that is normally done by the shell. Since Julia does this itself, it retains better control and can do some things that shells cannot.

Julia can run multiple commands in parallel:

```
julia> run(`echo hello` & `echo world`);
world
hello
```

The order of the output here is non-deterministic because the two `echo` processes are started nearly simultaneously, and race to make the first write to the `stdout` descriptor they share with each other and the `julia` parent process. Julia lets you pipe the output from both of these processes to another program:

```
julia> run(pipeline(`echo world` & `echo hello`, `sort`));
hello
world
```

In terms of UNIX plumbing, what's happening here is that a single UNIX pipe object is created and written to by both `echo` processes, and the other end of the pipe is read from by the `sort` command.

IO redirection can be accomplished by passing keyword arguments `stdin`, `stdout`, and `stderr` to the `pipeline` function:

```
pipeline(`do_work`, stdout=pipeline(`sort`, "out.txt"), stderr="errs.txt")
```

Avoiding Deadlock in Pipelines

When reading and writing to both ends of a pipeline from a single process, it is important to avoid forcing the kernel to buffer all of the data.

For example, when reading all of the output from a command, call `read(out, String)`, not `wait(process)`, since the former will actively consume all of the data written by the process, whereas the latter will attempt to store the data in the kernel's buffers while waiting for a reader to be connected.

Another common solution is to separate the reader and writer of the pipeline into separate `Task`s:

```
writer = Threads.@spawn write(process, "data")
reader = Threads.@spawn do_compute(read(process, String))
wait(writer)
fetch(reader)
```

(commonly also, reader is not a separate task, since we immediately `fetch` it anyways).

Complex Example

The combination of a high-level programming language, a first-class command abstraction, and automatic setup of pipes between processes is a powerful one. To give some sense of the complex pipelines that can be created easily, here are some more sophisticated examples, with apologies for the excessive use of Perl one-liners:

```
julia> prefixer(prefix, sleep) = `perl -nle '$|=1; print "'$prefix' ", $_; sleep '$sleep';'`;

julia> run(pipeline(`perl -le '$|=1; for(0..5){ print; sleep 1 }'`, prefixer("A",2) &
↪  prefixer("B",2)));
B 0
A 1
B 2
A 3
B 4
A 5
```

This is a classic example of a single producer feeding two concurrent consumers: one `perl` process generates lines with the numbers 0 through 5 on them, while two parallel processes consume that output, one prefixing lines with the letter "A", the other with the letter "B". Which consumer gets the first line is non-deterministic, but once that race has been won, the lines are consumed alternately by one process and then the other. (Setting `$|=1` in Perl causes each print statement to flush the `stdout` handle, which is necessary for this example to work. Otherwise all the output is buffered and printed to the pipe at once, to be read by just one consumer process.)

Here is an even more complex multi-stage producer-consumer example:

```
julia> run(pipeline(`perl -le '$|=1; for(0..5){ print; sleep 1 }'`,
           prefixer("X",3) & prefixer("Y",3) & prefixer("Z",3),
           prefixer("A",2) & prefixer("B",2)));
A X 0
B Y 1
A Z 2
B X 3
A Y 4
B Z 5
```

This example is similar to the previous one, except there are two stages of consumers, and the stages have different latency so they use a different number of parallel workers, to maintain saturated throughput.

We strongly encourage you to try all these examples to see how they work.

27.4 `Cmd` Objects

The backtick syntax create an object of type `Cmd`. Such object may also be constructed directly from an existing `Cmd` or list of arguments:

```
run(Cmd(`pwd`, dir=".."))
run(Cmd(["pwd"], detach=true, ignorestatus=true))
```

This allows you to specify several aspects of the `Cmd`'s execution environment via keyword arguments. For example, the `dir` keyword provides control over the `Cmd`'s working directory:

```
julia> run(Cmd(`pwd`, dir="/"));
/
```

And the env keyword allows you to set execution environment variables:

```
julia> run(Cmd(`sh -c "echo foo \$HOWLONG"`, env=("HOWLONG" => "ever!",)));
foo ever!
```

See `Cmd` for additional keyword arguments. The `setenv` and `addenv` commands provide another means for replacing or adding to the `Cmd` execution environment variables, respectively:

```
julia> run(setenv(`sh -c "echo foo \$HOWLONG"`, ("HOWLONG" => "ever!",)));
foo ever!

julia> run(addenv(`sh -c "echo foo \$HOWLONG"`, "HOWLONG" => "ever!"));
foo ever!
```

Chapter 28

Calling C and Fortran Code

Though most code can be written in Julia, there are many high-quality, mature libraries for numerical computing already written in C and Fortran. To allow easy use of this existing code, Julia makes it simple and efficient to call C and Fortran functions. Julia has a "no boilerplate" philosophy: functions can be called directly from Julia without any "glue" code, code generation, or compilation – even from the interactive prompt. This is accomplished just by making an appropriate call with the `@ccall` macro (or the less convenient `ccall` syntax, see the `ccall` syntax section).

The code to be called must be available as a shared library. Most C and Fortran libraries ship compiled as shared libraries already, but if you are compiling the code yourself using GCC (or Clang), you will need to use the `-shared` and `-fPIC` options. The machine instructions generated by Julia's JIT are the same as a native C call would be, so the resulting overhead is the same as calling a library function from C code. [1]

By default, Fortran compilers generate mangled names (for example, converting function names to lowercase or uppercase, often appending an underscore), and so to call a Fortran function you must pass the mangled identifier corresponding to the rule followed by your Fortran compiler. Also, when calling a Fortran function, all inputs must be passed as pointers to allocated values on the heap or stack. This applies not only to arrays and other mutable objects which are normally heap-allocated, but also to scalar values such as integers and floats which are normally stack-allocated and commonly passed in registers when using C or Julia calling conventions.

The syntax for `@ccall` to generate a call to the library function is:

```
@ccall library.function_name(argvalue1::argtype1, ...)::returntype
@ccall function_name(argvalue1::argtype1, ...)::returntype
@ccall $function_pointer(argvalue1::argtype1, ...)::returntype
```

where `library` is a string constant or global variable name (see Non-constant Function -Specifications below). The library can be just a name, or it can specify a full path to the library. The library may be omitted, in which case the function name is resolved in the current executable, the current libc, or libjulia(-internal). This form can be used to call C library functions, functions in the Julia runtime, or functions in an application linked to Julia. Omitting the library *cannot* be used to call a function in any library (like specifying `RTLD_DEFAULT` to `dlsym`) as such behavior is slow, complicated, and not implemented on all platforms. Alternatively, `@ccall` may also be used to call a function pointer `$function_pointer`, such as one returned by `Libdl.dlsym`. The `argtypes` corresponds to the C-function signature and the `argvalues` are the actual argument values to be passed to the function.

Note

See below for how to map C types to Julia types.

As a complete but simple example, the following calls the `clock` function from the standard C library on most Unix-derived systems:

```
julia> t = @ccall clock()::Int32
2292761

julia> typeof(t)
Int32
```

`clock` takes no arguments and returns an `Int32`. To call the `getenv` function to get a pointer to the value of an environment variable, one makes a call like this:

```
julia> path = @ccall getenv("SHELL"::Cstring)::Cstring
Cstring(@0x00007fff5fbffc45)

julia> unsafe_string(path)
"/bin/bash"
```

In practice, especially when providing reusable functionality, one generally wraps `@ccall` uses in Julia functions that set up arguments and then check for errors in whatever manner the C or Fortran function specifies. If an error occurs it is thrown as a normal Julia exception. This is especially important since C and Fortran APIs are notoriously inconsistent about how they indicate error conditions. For example, the `getenv` C library function is wrapped in the following Julia function, which is a simplified version of the actual definition from `env.jl`:

```
function getenv(var::AbstractString)
    val = @ccall getenv(var::Cstring)::Cstring
    if val == C_NULL
        error("getenv: undefined variable: ", var)
    end
    return unsafe_string(val)
end
```

The C `getenv` function indicates an error by returning `C_NULL`, but other standard C functions indicate errors in different ways, including by returning -1, 0, 1, and other special values. This wrapper throws an exception indicating the problem if the caller tries to get a non-existent environment variable:

```
julia> getenv("SHELL")
"/bin/bash"

julia> getenv("FOOBAR")
ERROR: getenv: undefined variable: FOOBAR
```

Here is a slightly more complex example that discovers the local machine's hostname.

```
function gethostname()
    hostname = Vector{UInt8}(undef, 256) # MAXHOSTNAMELEN
    err = @ccall gethostname(hostname::Ptr{UInt8}, sizeof(hostname)::Csize_t)::Int32
    Base.systemerror("gethostname", err != 0)
    hostname[end] = 0 # ensure null-termination
    return GC.@preserve hostname unsafe_string(pointer(hostname))
end
```

This example first allocates an array of bytes. It then calls the C library function `gethostname` to populate the array with the hostname. Finally, it takes a pointer to the hostname buffer, and converts the pointer to a Julia string, assuming that it is a null-terminated C string.

It is common for C libraries to use this pattern of requiring the caller to allocate memory to be passed to the callee and populated. Allocation of memory from Julia like this is generally accomplished by creating an uninitialized array and passing a pointer to its data to the C function. This is why we don't use the `Cstring` type here: as the array is uninitialized, it could contain null bytes. Converting to a `Cstring` as part of the `@ccall` checks for contained null bytes and could therefore throw a conversion error.

Dereferencing `pointer(hostname)` with `unsafe_string` is an unsafe operation as it requires access to the memory allocated for `hostname` that may have been in the meanwhile garbage collected. The macro `GC.@preserve` prevents this from happening and therefore accessing an invalid memory location.

Finally, here is an example of specifying a library via a path. We create a shared library with the following content

```
#include <stdio.h>

void say_y(int y)
{
    printf("Hello from C: got y = %d.\n", y);
}
```

and compile it with `gcc -fPIC -shared -o mylib.so mylib.c`. It can then be called by specifying the (absolute) path as the library name:

```
julia> @ccall "./mylib.so".say_y(5::Cint)::Cvoid
Hello from C: got y = 5.
```

28.1 Creating C-Compatible Julia Function Pointers

It is possible to pass Julia functions to native C functions that accept function pointer arguments. For example, to match C prototypes of the form:

```
typedef returntype (*functiontype)(argumenttype, ...)
```

The macro `@cfunction` generates the C-compatible function pointer for a call to a Julia function. The arguments to `@cfunction` are:

1. A Julia function

2. The function's return type
3. A tuple of input types, corresponding to the function signature

Note

As with `@ccall`, the return type and the input types must be literal constants.

Note

Currently, only the platform-default C calling convention is supported. This means that `@cfunction`-generated pointers cannot be used in calls where WINAPI expects a `stdcall` function on 32-bit Windows, but can be used on WIN64 (where `stdcall` is unified with the C calling convention).

Note

Callback functions exposed via `@cfunction` should not throw errors, as that will return control to the Julia runtime unexpectedly and may leave the program in an undefined state.

A classic example is the standard C library `qsort` function, declared as:

```
void qsort(void *base, size_t nitems, size_t size,
           int (*compare)(const void*, const void*));
```

The `base` argument is a pointer to an array of length `nitems`, with elements of `size` bytes each. `compare` is a callback function which takes pointers to two elements `a` and `b` and returns an integer less/greater than zero if `a` should appear before/after `b` (or zero if any order is permitted).

Now, suppose that we have a 1-d array `A` of values in Julia that we want to sort using the `qsort` function (rather than Julia's built-in `sort` function). Before we consider calling `qsort` and passing arguments, we need to write a comparison function:

```
julia> function mycompare(a, b)::Cint
           return (a < b) ? -1 : ((a > b) ? +1 : 0)
       end;
```

`qsort` expects a comparison function that return a C `int`, so we annotate the return type to be `Cint`.

In order to pass this function to C, we obtain its address using the macro `@cfunction`:

```
julia> mycompare_c = @cfunction(mycompare, Cint, (Ref{Cdouble}, Ref{Cdouble}));
```

`@cfunction` requires three arguments: the Julia function (`mycompare`), the return type (`Cint`), and a literal tuple of the input argument types, in this case to sort an array of `Cdouble` (`Float64`) elements.

The final call to `qsort` looks like this:

```
julia> A = [1.3, -2.7, 4.4, 3.1];

julia> @ccall qsort(A::Ptr{Cdouble}, length(A)::Csize_t, sizeof(eltype(A))::Csize_t,
↪  mycompare_c::Ptr{Cvoid})::Cvoid

julia> A
4-element Vector{Float64}:
 -2.7
  1.3
  3.1
  4.4
```

As the example shows, the original Julia array `A` has now been sorted: `[-2.7, 1.3, 3.1, 4.4]`. Note that Julia takes care of converting the array to a `Ptr{Cdouble}`, computing the size of the element type in bytes, and so on.

For fun, try inserting a `println("mycompare($a, $b)")` line into `mycompare`, which will allow you to see the comparisons that `qsort` is performing (and to verify that it is really calling the Julia function that you passed to it).

28.2 Mapping C Types to Julia

It is critical to exactly match the declared C type with its declaration in Julia. Inconsistencies can cause code that works correctly on one system to fail or produce indeterminate results on a different system.

Note that no C header files are used anywhere in the process of calling C functions: you are responsible for making sure that your Julia types and call signatures accurately reflect those in the C header file.[2]

Automatic Type Conversion

Julia automatically inserts calls to the `Base.cconvert` function to convert each argument to the specified type. For example, the following call:

```
@ccall "libfoo".foo(x::Int32, y::Float64)::Cvoid
```

will behave as if it were written like this:

```
c_x = Base.cconvert(Int32, x)
c_y = Base.cconvert(Float64, y)
GC.@preserve c_x c_y begin
    @ccall "libfoo".foo(
        Base.unsafe_convert(Int32, c_x)::Int32,
        Base.unsafe_convert(Float64, c_y)::Float64
    )::Cvoid
end
```

`Base.cconvert` normally just calls `convert`, but can be defined to return an arbitrary new object more appropriate for passing to C. This should be used to perform all allocations of memory that will be accessed by the C code. For example, this is used to convert an `Array` of objects (e.g. strings) to an array of pointers.

`Base.unsafe_convert` handles conversion to `Ptr` types. It is considered unsafe because converting an object to a native pointer can hide the object from the garbage collector, causing it to be freed prematurely.

Type Correspondences

First, let's review some relevant Julia type terminology:

Syntax / Keyword	Example	Description
`mutable struct`	`BitSet`	"Concrete Type" :: A group of related data that includes a type-tag, is managed by the Julia GC, and is defined by object-identity. The type parameters of a concrete type must be fully defined (no `TypeVars` are allowed) in order for the instance to be constructed. Also see `isconcretetype`.
`abstract type`	`Any`, `AbstractArray{T, N}`, `Complex{T}`	"Super Type" :: A super-type (not a concrete type) that cannot be instantiated, but can be used to describe a group of types. Also see `isabstracttype`.
`T{A}`	`Vector{Int}`	"Type Parameter" :: A specialization of a type (typically used for dispatch or storage optimization). "TypeVar" :: The T in the type parameter declaration is referred to as a TypeVar (short for type variable).
`primitive type`	`Int`, `Float64`	"Primitive Type" :: A type with no fields, but a size. It is stored and defined by-value.
`struct`	`Pair{Int, Int}`	"Struct" :: A type with all fields defined to be constant. It is defined by-value, and may be stored with a type-tag.
	`ComplexF64 (isbits)`	"Is-Bits" :: A `primitive type`, or a `struct` type where all fields are other `isbits` types. It is defined by-value, and is stored without a type-tag.
`struct ...; end`	`nothing`	"Singleton" :: a concrete Type or Struct with no fields.
`(...)` or `tuple(...)`	`(1, 2, 3)`	"Tuple" :: an immutable data-structure similar to an anonymous struct type, or a constant array. Represented as either an array or a struct.

Bits Types

There are several special types to be aware of, as no other type can be defined to behave the same:

- `Float32`

  Exactly corresponds to the `float` type in C (or `REAL*4` in Fortran).

- `Float64`

  Exactly corresponds to the `double` type in C (or `REAL*8` in Fortran).

- `ComplexF32`

  Exactly corresponds to the `complex float` type in C (or `COMPLEX*8` in Fortran).

- `ComplexF64`

  Exactly corresponds to the `complex double` type in C (or `COMPLEX*16` in Fortran).

- `Signed`

  Exactly corresponds to the `signed` type annotation in C (or any `INTEGER` type in Fortran). Any Julia type that is not a subtype of `Signed` is assumed to be unsigned.

- `Ref{T}`

  Behaves like a `Ptr{T}` that can manage its memory via the Julia GC.

- `Array{T,N}`

  When an array is passed to C as a `Ptr{T}` argument, it is not reinterpret-cast: Julia requires that the element type of the array matches T, and the address of the first element is passed.

  Therefore, if an `Array` contains data in the wrong format, it will have to be explicitly converted using a call such as `trunc.(Int32, A)`.

  To pass an array `A` as a pointer of a different type *without* converting the data beforehand (for example, to pass a `Float64` array to a function that operates on uninterpreted bytes), you can declare the argument as `Ptr{Cvoid}`.

  If an array of eltype `Ptr{T}` is passed as a `Ptr{Ptr{T}}` argument, `Base.cconvert` will attempt to first make a null-terminated copy of the array with each element replaced by its `Base.cconvert` version. This allows, for example, passing an `argv` pointer array of type `Vector{String}` to an argument of type `Ptr{Ptr{Cchar}}`.

On all systems we currently support, basic C/C++ value types may be translated to Julia types as follows. Every C type also has a corresponding Julia type with the same name, prefixed by C. This can help when writing portable code (and remembering that an `int` in C is not the same as an `Int` in Julia).

System Independent Types

The `Cstring` type is essentially a synonym for `Ptr{UInt8}`, except the conversion to `Cstring` throws an error if the Julia string contains any embedded null characters (which would cause the string to be silently truncated if the C routine treats null as the terminator). If you are passing a `char*` to a C routine that does not assume null termination (e.g. because you pass an explicit string length), or if you know for certain that your Julia string does not contain null and want to skip the check, you can use `Ptr{UInt8}` as the argument type. `Cstring` can also be used as the `ccall` return type, but in that case it obviously does not introduce any extra checks and is only meant to improve the readability of the call.

System Dependent Types

Note

When calling Fortran, all inputs must be passed by pointers to heap- or stack-allocated values, so all type correspondences above should contain an additional `Ptr{..}` or `Ref{..}` wrapper around their type specification.

Warning

For string arguments (`char*`) the Julia type should be `Cstring` (if null-terminated data is expected), or either `Ptr{Cchar}` or `Ptr{UInt8}` otherwise (these two pointer types have the same effect), as described above, not `String`. Similarly, for array arguments (`T[]` or `T*`), the Julia type should again be `Ptr{T}`, not `Vector{T}`.

Warning

Julia's `Char` type is 32 bits, which is not the same as the wide-character type (`wchar_t` or `wint_t`) on all platforms.

Warning

A return type of `Union{}` means the function will not return, i.e., C++11 `[[noreturn]]` or C11 `_Noreturn` (e.g. `jl_throw` or `longjmp`). Do not use this for functions that return no value (`void`) but do return, for those, use `Cvoid` instead.

Note

For `wchar_t*` arguments, the Julia type should be `Cwstring` (if the C routine expects a null-terminated string), or `Ptr{Cwchar_t}` otherwise. Note also that UTF-8 string data in Julia is internally null-terminated, so it can be passed to C functions expecting null-terminated data without making a copy (but using the `Cwstring` type will cause an error to be thrown if the string itself contains null characters).

Note

C functions that take an argument of type `char**` can be called by using a `Ptr{Ptr{UInt8}}` type within Julia. For example, C functions of the form:

```
int main(int argc, char **argv);
```

can be called via the following Julia code:

```
argv = [ "a.out", "arg1", "arg2" ]
@ccall main(length(argv)::Int32, argv::Ptr{Ptr{UInt8}})::Int32
```

Note

For Fortran functions taking variable length strings of type `character(len=*)` the string lengths are provided as *hidden arguments*. Type and position of these arguments in the list are compiler specific, where compiler vendors usually default to using `Csize_t` as type and append the hidden arguments at the end of the argument list. While this behaviour is fixed for some compilers (GNU), others *optionally* permit placing hidden arguments directly after the character argument (Intel, PGI). For example, Fortran subroutines of the form

```
subroutine test(str1, str2)
character(len=*) :: str1,str2
```

can be called via the following Julia code, where the lengths are appended

```
str1 = "foo"
str2 = "bar"
ccall(:test, Cvoid, (Ptr{UInt8}, Ptr{UInt8}, Csize_t, Csize_t),
                    str1, str2, sizeof(str1), sizeof(str2))
```

Warning

Fortran compilers *may* also add other hidden arguments for pointers, assumed-shape `(:)` and assumed-size `(*)` arrays. Such behaviour can be avoided by using `ISO_C_BINDING` and including `bind(c)` in the definition of the subroutine, which is strongly recommended for interoperable code. In this case, there will be no hidden arguments, at the cost of some language features (e.g. only `character(len=1)` will be permitted to pass strings).

Note

A C function declared to return `Cvoid` will return the value `nothing` in Julia.

Struct Type Correspondences

Composite types such as `struct` in C or TYPE in Fortran90 (or STRUCTURE / RECORD in some variants of F77), can be mirrored in Julia by creating a `struct` definition with the same field layout.

When used recursively, `isbits` types are stored inline. All other types are stored as a pointer to the data. When mirroring a struct used by-value inside another struct in C, it is imperative that you do not attempt to manually copy the fields over, as this will not preserve the correct field alignment. Instead, declare an `isbits` struct type and use that instead. Unnamed structs are not possible in the translation to Julia.

Packed structs and union declarations are not supported by Julia.

You can get an approximation of a `union` if you know, a priori, the field that will have the greatest size (potentially including padding). When translating your fields to Julia, declare the Julia field to be only of that type.

Arrays of parameters can be expressed with `NTuple`. For example, the struct in C notation is written as

```
struct B {
```

```
    int A[3];
};

b_a_2 = B.A[2];
```

can be written in Julia as

```
struct B
    A::NTuple{3, Cint}
end

b_a_2 = B.A[3]  # note the difference in indexing (1-based in Julia, 0-based in C)
```

Arrays of unknown size (C99-compliant variable length structs specified by `[]` or `[0]`) are not directly supported. Often the best way to deal with these is to deal with the byte offsets directly. For example, if a C library declared a proper string type and returned a pointer to it:

```
struct String {
    int strlen;
    char data[];
};
```

In Julia, we can access the parts independently to make a copy of that string:

```
str = from_c::Ptr{Cvoid}
len = unsafe_load(Ptr{Cint}(str))
unsafe_string(str + Core.sizeof(Cint), len)
```

Type Parameters

The type arguments to `@ccall` and `@cfunction` are evaluated statically, when the method containing the usage is defined. They therefore must take the form of a literal tuple, not a variable, and cannot reference local variables.

This may sound like a strange restriction, but remember that since C is not a dynamic language like Julia, its functions can only accept argument types with a statically-known, fixed signature.

However, while the type layout must be known statically to compute the intended C ABI, the static parameters of the function are considered to be part of this static environment. The static parameters of the function may be used as type parameters in the call signature, as long as they don't affect the layout of the type. For example, `f(x::T) where {T} = @ccall valid(x::Ptr{T})::Ptr{T}` is valid, since `Ptr` is always a word-size primitive type. But, `g(x::T) where {T} = @ccall notvalid(x::T)::T` is not valid, since the type layout of T is not known statically.

SIMD Values

If a C/C++ routine has an argument or return value that is a native SIMD type, the corresponding Julia type is a homogeneous tuple of `VecElement` that naturally maps to the SIMD type. Specifically:

- The tuple must be the same size and elements as the SIMD type. For example, a tuple representing an __m128 on x86 must have a size of 16 bytes and Float32 elements.
- The element type of the tuple must be an instance of `VecElement{T}` where T is a primitive type with a power-of-two number of bytes (e.g. 1, 2, 4, 8, 16, etc) such as Int8 or Float64.

For instance, consider this C routine that uses AVX intrinsics:

```
#include <immintrin.h>

__m256 dist( __m256 a, __m256 b ) {
    return _mm256_sqrt_ps(_mm256_add_ps(_mm256_mul_ps(a, a),
                                        _mm256_mul_ps(b, b)));
}
```

The following Julia code calls `dist` using `ccall`:

```
const m256 = NTuple{8, VecElement{Float32}}

a = m256(ntuple(i -> VecElement(sin(Float32(i))), 8))
b = m256(ntuple(i -> VecElement(cos(Float32(i))), 8))

function call_dist(a::m256, b::m256)
    @ccall "libdist".dist(a::m256, b::m256)::m256
end

println(call_dist(a,b))
```

The host machine must have the requisite SIMD registers. For example, the code above will not work on hosts without AVX support.

Memory Ownership

`malloc/free`

Memory allocation and deallocation of such objects must be handled by calls to the appropriate cleanup routines in the libraries being used, just like in any C program. Do not try to free an object received from a C library with `Libc.free` in Julia, as this may result in the `free` function being called via the wrong library and cause the process to abort. The reverse (passing an object allocated in Julia to be freed by an external library) is equally invalid.

When to use `T`, `Ptr{T}` and `Ref{T}`

In Julia code wrapping calls to external C routines, ordinary (non-pointer) data should be declared to be of type T inside the `@ccall`, as they are passed by value. For C code accepting pointers, `Ref{T}` should generally be used for the types of input arguments, allowing the use of pointers to memory managed by either Julia or C through the implicit call to `Base.cconvert`. In contrast, pointers returned by the C function called should be declared to be of the output type `Ptr{T}`, reflecting that the memory pointed to is managed by C only. Pointers contained in C structs should be represented as fields of type `Ptr{T}` within the corresponding Julia struct types designed to mimic the internal structure of corresponding C structs.

In Julia code wrapping calls to external Fortran routines, all input arguments should be declared as of type `Ref{T}`, as Fortran passes all variables by pointers to memory locations. The return type should either be `Cvoid` for Fortran subroutines, or a T for Fortran functions returning the type T.

28.3 Mapping C Functions to Julia

`@ccall` / `@cfunction` argument translation guide

For translating a C argument list to Julia:

- T, where T is one of the primitive types: `char`, `int`, `long`, `short`, `float`, `double`, `complex`, `enum` or any of their `typedef` equivalents
  - T, where T is an equivalent Julia Bits Type (per the table above)
  - if T is an `enum`, the argument type should be equivalent to `Cint` or `Cuint`
  - argument value will be copied (passed by value)
- `struct T` (including typedef to a struct)
  - T, where T is a concrete Julia type
  - argument value will be copied (passed by value)
- `vector T` (or `__attribute__ vector_size`, or a typedef such as `__m128`)
  - `NTuple{N, VecElement{T}}`, where T is a primitive Julia type of the correct size and N is the number of elements in the vector (equal to `vector_size / sizeof T`).
- `void*`
  - depends on how this parameter is used, first translate this to the intended pointer type, then determine the Julia equivalent using the remaining rules in this list
  - this argument may be declared as `Ptr{Cvoid}` if it really is just an unknown pointer
- `jl_value_t*`
  - `Any`
  - argument value must be a valid Julia object
- `jl_value_t* const*`
  - `Ref{Any}`
  - argument list must be a valid Julia object (or `C_NULL`)
  - cannot be used for an output parameter, unless the user is able to separately arrange for the object to be GC-preserved
- `T*`
  - `Ref{T}`, where T is the Julia type corresponding to T

    - argument value will be copied if it is an `inlinealloc` type (which includes `isbits` otherwise, the value must be a valid Julia object

- `T (*)(...)` (e.g. a pointer to a function)

    - `Ptr{Cvoid}` (you may need to use `@cfunction` explicitly to create this pointer)

- `...` (e.g. a vararg)

for `ccall` : `T...`, where T is the single Julia type of all remaining arguments

for `@ccall` : `; va_arg1::T, va_arg2::S, etc`, where T and S are the Julia type (i.e. separate the regular arguments from varargs with a `;`)

    - currently unsupported by `@cfunction`

- `va_arg`

    - not supported by `ccall` or `@cfunction`

`@ccall` / `@cfunction` return type translation guide

For translating a C return type to Julia:

- `void`

    - `Cvoid` (this will return the singleton instance `nothing::Cvoid`)

- T, where T is one of the primitive types: `char`, `int`, `long`, `short`, `float`, `double`, `complex`, `enum` or any of their `typedef` equivalents

    - same as C argument list
    - argument value will be copied (returned by-value)

- `struct T` (including typedef to a struct)

    - same as C argument list
    - argument value will be copied (returned by-value)

- `vector T`

    - same as C argument list

- `void*`

    - depends on how this parameter is used, first translate this to the intended pointer type, then determine the Julia equivalent using the remaining rules in this list
    - this argument may be declared as `Ptr{Cvoid}` if it really is just an unknown pointer

- `jl_value_t*`

    - `Any`

  - argument value must be a valid Julia object

- `jl_value_t**`
  - `Ptr{Any}` (`Ref{Any}` is invalid as a return type)
- `T*`
  - If the memory is already owned by Julia, or is an `isbits` type, and is known to be non-null:
    - `Ref{T}`, where T is the Julia type corresponding to T
    - a return type of `Ref{Any}` is invalid, it should either be `Any` (corresponding to `jl_value_t*`) or `Ptr{Any}` (corresponding to `jl_value_t**`)
    - C **MUST NOT** modify the memory returned via `Ref{T}` if T is an `isbits` type
  - If the memory is owned by C:
    - `Ptr{T}`, where T is the Julia type corresponding to T
- `T (*)(...)` (e.g. a pointer to a function)
  - `Ptr{Cvoid}` to call this directly from Julia you will need to pass this as the first argument to `@ccall`. See Indirect Calls.

Passing Pointers for Modifying Inputs

Because C doesn't support multiple return values, often C functions will take pointers to data that the function will modify. To accomplish this within a `@ccall`, you need to first encapsulate the value inside a `Ref{T}` of the appropriate type. When you pass this `Ref` object as an argument, Julia will automatically pass a C pointer to the encapsulated data:

```
width = Ref{Cint}(0)
range = Ref{Cfloat}(0)
@ccall foo(width::Ref{Cint}, range::Ref{Cfloat})::Cvoid
```

Upon return, the contents of `width` and `range` can be retrieved (if they were changed by `foo`) by `width[]` and `range[]`; that is, they act like zero-dimensional arrays.

28.4 C Wrapper Examples

Let's start with a simple example of a C wrapper that returns a `Ptr` type:

```
mutable struct gsl_permutation
end

# The corresponding C signature is
#     gsl_permutation * gsl_permutation_alloc (size_t n);
function permutation_alloc(n::Integer)
    output_ptr = @ccall "libgsl".gsl_permutation_alloc(n::Csize_t)::Ptr{gsl_permutation}
    if output_ptr == C_NULL # Could not allocate memory
        throw(OutOfMemoryError())
    end
    return output_ptr
end
```

The GNU Scientific Library (here assumed to be accessible through `:libgsl`) defines an opaque pointer, `gsl_permutation *`, as the return type of the C function `gsl_permutation_alloc`. As user code never has to look inside the `gsl_permutation` struct, the corresponding Julia wrapper simply needs a new type declaration, `gsl_permutation`, that has no internal fields and whose sole purpose is to be placed in the type parameter of a `Ptr` type. The return type of the `ccall` is declared as `Ptr{gsl_permutation}`, since the memory allocated and pointed to by `output_ptr` is controlled by C.

The input `n` is passed by value, and so the function's input signature is simply declared as `::Csize_t` without any `Ref` or `Ptr` necessary. (If the wrapper was calling a Fortran function instead, the corresponding function input signature would instead be `::Ref{Csize_t}`, since Fortran variables are passed by pointers.) Furthermore, `n` can be any type that is convertible to a `Csize_t` integer; the `ccall` implicitly calls `Base.cconvert(Csize_t, n)`.

Here is a second example wrapping the corresponding destructor:

```
# The corresponding C signature is
#     void gsl_permutation_free (gsl_permutation * p);
function permutation_free(p::Ptr{gsl_permutation})
    @ccall "libgsl".gsl_permutation_free(p::Ptr{gsl_permutation})::Cvoid
end
```

Here is a third example passing Julia arrays:

```
# The corresponding C signature is
#    int gsl_sf_bessel_Jn_array (int nmin, int nmax, double x,
#                                double result_array[])
function sf_bessel_Jn_array(nmin::Integer, nmax::Integer, x::Real)
    if nmax < nmin
        throw(DomainError())
    end
    result_array = Vector{Cdouble}(undef, nmax - nmin + 1)
    errorcode = @ccall "libgsl".gsl_sf_bessel_Jn_array(
                    nmin::Cint, nmax::Cint, x::Cdouble, result_array::Ref{Cdouble})::Cint
    if errorcode != 0
        error("GSL error code $errorcode")
    end
    return result_array
end
```

The C function wrapped returns an integer error code; the results of the actual evaluation of the Bessel J function populate the Julia array `result_array`. This variable is declared as a `Ref{Cdouble}`, since its memory is allocated and managed by Julia. The implicit call to `Base.cconvert(Ref{Cdouble}, result_array)` unpacks the Julia pointer to a Julia array data structure into a form understandable by C.

28.5 Fortran Wrapper Example

The following example utilizes `ccall` to call a function in a common Fortran library (libBLAS) to compute a dot product. Notice that the argument mapping is a bit different here than above, as we need to map from Julia to Fortran. On every argument type, we specify `Ref` or `Ptr`. This mangling convention may be specific to your Fortran compiler and operating system and is likely undocumented. However, wrapping each in a `Ref` (or `Ptr`, where equivalent) is a frequent requirement of Fortran compiler implementations:

```
function compute_dot(DX::Vector{Float64}, DY::Vector{Float64})
    @assert length(DX) == length(DY)
    n = length(DX)
    incx = incy = 1
    product = @ccall "libLAPACK".ddot(
        n::Ref{Int32}, DX::Ptr{Float64}, incx::Ref{Int32}, DY::Ptr{Float64},
        ↪  incy::Ref{Int32})::Float64
    return product
end
```

28.6 Garbage Collection Safety

When passing data to a `@ccall`, it is best to avoid using the `pointer` function. Instead define a `Base.cconvert` method and pass the variables directly to the `@ccall`. `@ccall` automatically arranges that all of its arguments will be preserved from garbage collection until the call returns. If a C API will store a reference to memory allocated by Julia, after the `@ccall` returns, you must ensure that the object remains visible to the garbage collector. The suggested way to do this is to make a global variable of type `Vector{Ref}` to hold these values until the C library notifies you that it is finished with them.

Whenever you have created a pointer to Julia data, you must ensure the original data exists until you have finished using the pointer. Many methods in Julia such as `unsafe_load` and `String` make copies of data instead of taking ownership of the buffer, so that it is safe to free (or alter) the original data without affecting Julia. A notable exception is `unsafe_wrap` which, for performance reasons, shares (or can be told to take ownership of) the underlying buffer.

The garbage collector does not guarantee any order of finalization. That is, if a contained a reference to b and both a and b are due for garbage collection, there is no guarantee that b would be finalized after a. If proper finalization of a depends on b being valid, it must be handled in other ways.

28.7 Non-constant Function Specifications

In some cases, the exact name or path of the needed library is not known in advance and must be computed at run time. To handle such cases, the library component specification can be a value such as `Libdl.LazyLibrary`. The runtime will call `Libdl.dlopen` on that object when first used by a `ccall`.

Using LazyLibrary for Lazy Loading

`Libdl.LazyLibrary` provides a thread-safe mechanism for deferring library loading until first use. This is the recommended approach for library initialization in modern Julia code.

A `LazyLibrary` represents a library that opens itself (and its dependencies) automatically on first use in a `ccall()`, `@ccall`, `dlopen()`, `dlsym()`, `dlpath()`, or `cglobal()`. The library is loaded exactly once in a thread-safe manner, and subsequent calls reuse the loaded library handle.

Basic Usage

```
using Libdl

# Define a LazyLibrary as a const for optimal performance
const libz = LazyLibrary("libz")

# Use directly in @ccall - library loads automatically on first call
```

```
@ccall libz.deflate(strm::Ptr{Cvoid}, flush::Cint)::Cint

# Also works with ccall
ccall((:inflate, libz), Cint, (Ptr{Cvoid}, Cint), strm, flush)
```

Platform-Specific Libraries

For code that needs to work across different platforms:

```
const mylib = LazyLibrary(
    if Sys.iswindows()
        "mylib.dll"
    elseif Sys.isapple()
        "libmylib.dylib"
    else
        "libmylib.so"
    end
)
```

Libraries with Dependencies

When a library depends on other libraries, specify the dependencies to ensure they load in the correct order:

```
const libfoo = LazyLibrary("libfoo")
const libbar = LazyLibrary("libbar"; dependencies=[libfoo])

# When libbar is first used, libfoo is loaded first automatically
@ccall libbar.bar_function(x::Cint)::Cint
```

Lazy Path Construction

For libraries whose paths are determined at runtime, use `LazyLibraryPath`:

```
# Path is constructed when library is first accessed
const mylib = LazyLibrary(LazyLibraryPath(artifact_dir, "lib", "libmylib.so"))
```

Initialization Callbacks

If a library requires initialization after loading:

```
const mylib = LazyLibrary("libmylib";
    on_load_callback = () -> @ccall mylib.initialize()::Cvoid
)
```

Warning

The `on_load_callback` should be minimal and must not call `wait()` on any tasks. It is called exactly once by the thread that loads the library.

Conversion from `__init__()` Pattern

Before `LazyLibrary`, library paths were often computed in `__init__()` functions. This pattern can be replaced with `LazyLibrary` for better performance and thread safety.

Old pattern using `__init__()`:

```
# Old: Library path computed in __init__()
libmylib_path = ""

function __init__(
    # Loads library on startup, whether it is used or not
    global libmylib_path = find_library(["libmylib"])
end

function myfunc(x)
    ccall((:cfunc, libmylib_path), Cint, (Cint,), x)
end
```

New pattern using `LazyLibrary`:

```
# New: Library as const, no __init__() needed
const libmylib = LazyLibrary("libmylib")

function myfunc(x)
    # Library loads automatically just before calling `cfunc`
    @ccall libmylib.cfunc(x::Cint)::Cint
end
```

For more details, see the `Libdl.LazyLibrary` documentation.

Overloading `dlopen` for Custom Types

The runtime will call `dlsym(:function, dlopen(library)::Ptr{Cvoid})` when a `@ccall` is executed. The `Libdl.dlopen` function can be overloaded for custom types to provide alternate behaviors. However, it is assumed that the library location and handle does not change once it is determined, so the result of the call may be cached and reused. Therefore, the number of times the `dlopen` expression executes is unspecified, and returning different values for multiple calls will results in unspecified (but valid) behavior.

Computed Function Names

If even more flexibility is needed, it is possible to use computed values as function names by staging through `eval` as follows:

```
@eval @ccall "lib".$(string("a", "b"))()::Cint
```

This expression constructs a name using `string`, then substitutes this name into a new `@ccall` expression, which is then evaluated. Keep in mind that `eval` only operates at the top level, so within this expression local variables will not be available (unless their values are substituted with `$`). For this reason, `eval` is typically only used to form top-level definitions, for example when wrapping libraries that contain many similar functions.

Indirect Calls

The first argument to `@ccall` can also be an expression to be evaluated at run time, each time it is used. In this case, the expression must evaluate to a `Ptr`, which will be used as the address of the native function to call. This behavior occurs when the first `@ccall` argument is marked with `$` and when the first `ccall` argument is not a simple constant literal or expression in `()`. The argument can be any expression and can use local variables and arguments and can return a different value every time.

For example, you might implement a macro similar to `cglobal` that looks up the function via `dlsym`, then caches the pointer in a shared reference (which is auto reset to C_NULL during precompile saving). For example:

```
macro dlsym(lib, func)
    z = Ref(C_NULL)
    quote
        local zlocal = $z[]
        if zlocal == C_NULL
            zlocal = dlsym($(esc(lib))::Ptr{Cvoid}, $(esc(func)))::Ptr{Cvoid}
            $z[] = zlocal
        end
        zlocal
    end
end

const mylibvar = LazyLibrary("mylib")
@ccall $(@dlsym(dlopen(mylibvar), "myfunc"))()::Cvoid
```

28.8 Closure cfunctions

The first argument to `@cfunction` can be marked with a `$`, in which case the return value will instead be a `struct CFunction` which closes over the argument. You must ensure that this return object is kept alive until all uses of it are done. The contents and code at the cfunction pointer will be erased via a `finalizer` when this reference is dropped and atexit. This is not usually needed, since this functionality is not present in C, but can be useful for dealing with ill-designed APIs which don't provide a separate closure environment parameter.

```
function qsort(a::Vector{T}, cmp) where T
    isbits(T) || throw(ArgumentError("this method can only qsort isbits arrays"))
    callback = @cfunction $cmp Cint (Ref{T}, Ref{T})
    # Here, `callback` isa Base.CFunction, which will be converted to Ptr{Cvoid}
    # (and protected against finalization) by the ccall
    @ccall qsort(a::Ptr{T}, length(a)::Csize_t, Base.elsize(a)::Csize_t, callback::Ptr{Cvoid})
    # We could instead use:
    #    GC.@preserve callback begin
    #        use(Base.unsafe_convert(Ptr{Cvoid}, callback))
    #    end
    # if we needed to use it outside of a `ccall`
    return a
end
```

Note

Closure `@cfunction` relies on LLVM trampolines, which are not available on all platforms (for example ARM and PowerPC).

28.9 Closing a Library

It is sometimes useful to close (unload) a library so that it can be reloaded. For instance, when developing C code for use with Julia, one may need to compile, call the C code from Julia, then close the library, make an edit, recompile, and load in the new changes. One can either restart Julia or use the `Libdl` functions to manage the library explicitly, such as:

```
lib = Libdl.dlopen("./my_lib.so") # Open the library explicitly.
sym = Libdl.dlsym(lib, :my_fcn)   # Get a symbol for the function to call.
@ccall $sym(...) # Use the pointer `sym` instead of the library.symbol tuple.
Libdl.dlclose(lib) # Close the library explicitly.
```

Note that when using `@ccall` with the input (e.g., `@ccall "./my_lib.so".my_fcn(...)::Cvoid`), the library is opened implicitly and it may not be explicitly closed.

28.10 Variadic function calls

To call variadic C functions a `semicolon` can be used in the argument list to separate required arguments from variadic arguments. An example with the `printf` function is given below:

```
julia> @ccall printf("%s = %d\n"::Cstring ; "foo"::Cstring, foo::Cint)::Cint
foo = 3
8
```

28.11 `ccall` interface

There is another alternative interface to `@ccall`. This interface is slightly less convenient but it does allow one to specify a calling convention.

The arguments to `ccall` are:

1. A `(:function, "library")` pair (most common),

   OR

   a `:function` name symbol or `"function"` name string (for symbols in the current process or libc),

   OR

   a function pointer (for example, from `dlsym`).

2. The function's return type

3. A tuple of input types, corresponding to the function signature. One common mistake is forgetting that a 1-tuple of argument types must be written with a trailing comma.

4. The actual argument values to be passed to the function, if any; each is a separate parameter.

Note

The `(:function, "library")` pair and the input type list must be syntactic tuples (i.e., they can't be variables or values with a type of Tuple.

The rettype and argument type values are evaluated at when the containing method is defined, not runtime.

Function Name vs Pointer Syntax

The syntax of the first argument to `ccall` determines whether you're calling by **name** or by **pointer**:

- **Name-based calls** (tuple literal syntax):
- Both the function and library names can be a quoted Symbol, a String, a variable name (a GlobalRef), or a dotted expression ending with a variable name.
- Single name: `(:function_name,)` or `"function_name"` - uses default library lookup.
- Name with library: `(:function_name, "library")` - specifies both function and library.
- Symbol, string, and tuple literal constants (not expressions that evaluate to those constants, but actual literals) are automatically normalized to tuple form.
- **Pointer-based calls** (non-tuple syntax):
- Anything that is not a literal tuple expression specified above is assumed to be an expression that evaluates to a function pointers at runtime.
- Function pointer variables: `fptr` where `fptr` is a runtime pointer value.
- Function pointer computations: `dlsym(:something)` where the result is computed at runtime every time (usually along with some caching logic).
- **Library name expressions**:
- When given as a variable, the library name can resolve to a `Symbol`, a `String`, or any other value. The runtime will call `Libdl.dlopen(name)` on the value an unspecified number of times, caching the result. The result is not invalidated if the value of the binding changes or if it becomes undefined, as long as there exists any value for that binding in any past or future worlds, that value may be used.
- Dot expressions, such as `A.B().c`, will be executed at method definition time up to the final `c`. The first part must resolve to a Module, and the second part to a quoted symbol. The value of that global will be resolved at runtime when the `ccall` is first executed.

A table of translations between the macro and function interfaces is given below.

28.12 Calling Convention

The second argument to `ccall` (immediately preceding return type) can optionally be a calling convention specifier (the `@ccall` macro currently does not support giving a calling convention). Without any specifier,

the platform-default C calling convention is used. Other supported conventions are: `stdcall`, `cdecl`, `fastcall`, and `thiscall` (no-op on 64-bit Windows). For example (from `base/libc.jl`) we see the same `gethostname` `ccall` as above, but with the correct signature for Windows:

```
hn = Vector{UInt8}(undef, 256)
err = ccall(:gethostname, stdcall, Int32, (Ptr{UInt8}, UInt32), hn, length(hn))
```

For more information, please see the LLVM Language Reference.

There is one additional special calling convention `llvmcall`, which allows inserting calls to LLVM intrinsics directly. This can be especially useful when targeting unusual platforms such as GPGPUs. For example, for CUDA, we need to be able to read the thread index:

```
ccall("llvm.nvvm.read.ptx.sreg.tid.x", llvmcall, Int32, ())
```

As with any `ccall`, it is essential to get the argument signature exactly correct. Also, note that there is no compatibility layer that ensures the intrinsic makes sense and works on the current target, unlike the equivalent Julia functions exposed by `Core.Intrinsics`.

28.13 Accessing Global Variables

Global variables exported by native libraries can be accessed by name using the `cglobal` function. The arguments to `cglobal` are a symbol specification identical to that used by `ccall`, and a type describing the value stored in the variable:

```
julia> cglobal((:errno, :libc), Int32)
Ptr{Int32} @0x00007f418d0816b8
```

The result is a pointer giving the address of the value. The value can be manipulated through this pointer using `unsafe_load` and `unsafe_store!`.

Note

This `errno` symbol may not be found in a library named "libc", as this is an implementation detail of your system compiler. Typically standard library symbols should be accessed just by name, allowing the compiler to fill in the correct one. Also, however, the `errno` symbol shown in this example is special in most compilers, and so the value seen here is probably not what you expect or want. Compiling the equivalent code in C on any multi-threaded-capable system would typically actually call a different function (via macro preprocessor overloading), and may give a different result than the legacy value printed here.

28.14 Accessing Data through a Pointer

The following methods are described as "unsafe" because a bad pointer or type declaration can cause Julia to terminate abruptly.

Given a `Ptr{T}`, the contents of type T can generally be copied from the referenced memory into a Julia object using `unsafe_load(ptr, [index])`. The index argument is optional (default is 1), and follows the

Julia-convention of 1-based indexing. This function is intentionally similar to the behavior of `getindex` and `setindex!` (e.g. `[]` access syntax).

The return value will be a new object initialized to contain a copy of the contents of the referenced memory. The referenced memory can safely be freed or released.

If T is Any, then the memory is assumed to contain a reference to a Julia object (a `jl_value_t*`), the result will be a reference to this object, and the object will not be copied. You must be careful in this case to ensure that the object was always visible to the garbage collector (pointers do not count, but the new reference does) to ensure the memory is not prematurely freed. Note that if the object was not originally allocated by Julia, the new object will never be finalized by Julia's garbage collector. If the `Ptr` itself is actually a `jl_value_t*`, it can be converted back to a Julia object reference by `unsafe_pointer_to_objref(ptr)`. (Julia values v can be converted to `jl_value_t*` pointers, as `Ptr{Cvoid}`, by calling `pointer_from_objref(v)`.)

The reverse operation (writing data to a `Ptr{T}`), can be performed using `unsafe_store!(ptr, value, [index])`. Currently, this is only supported for primitive types or other pointer-free (`isbits`) immutable struct types.

Any operation that throws an error is probably currently unimplemented and should be posted as a bug so that it can be resolved.

If the pointer of interest is a plain-data array (primitive type or immutable struct), the function `unsafe_wrap(Array, ptr,dims, own = false)` may be more useful. The final parameter should be true if Julia should "take ownership" of the underlying buffer and call `free(ptr)` when the returned `Array` object is finalized. If the own parameter is omitted or false, the caller must ensure the buffer remains in existence until all access is complete.

Arithmetic on the `Ptr` type in Julia (e.g. using +) does not behave the same as C's pointer arithmetic. Adding an integer to a `Ptr` in Julia always moves the pointer by some number of *bytes*, not elements. This way, the address values obtained from pointer arithmetic do not depend on the element types of pointers.

28.15 Thread-safety

Some C libraries execute their callbacks from a different thread, and since Julia isn't thread-safe you'll need to take some extra precautions. In particular, you'll need to set up a two-layered system: the C callback should only *schedule* (via Julia's event loop) the execution of your "real" callback. To do this, create an `AsyncCondition` object and `wait` on it:

```
cond = Base.AsyncCondition()
wait(cond)
```

The callback you pass to C should only execute a `ccall` to `:uv_async_send`, passing `cond.handle` as the argument, taking care to avoid any allocations or other interactions with the Julia runtime.

Note that events may be coalesced, so multiple calls to uv_async_send may result in a single wakeup notification to the condition.

28.16 More About Callbacks

For more details on how to pass callbacks to C libraries, see this blog post.

28.17 C++

For tools to create C++ bindings, see the CxxWrap package.

[1] Non-library function calls in both C and Julia can be inlined and thus may have even less overhead than calls to shared library functions. The point above is that the cost of actually doing foreign function call is about the same as doing a call in either native language.

[2] The Clang package can be used to auto-generate Julia code from a C header file.

C name	Fortran name	Standard Julia Alias	Julia Base Type
`unsigned char`	`CHARACTER`	`Cuchar`	`UInt8`
`bool` (`_Bool` in C99+)		`Cuchar`	`UInt8`
`short`	`INTEGER*2`, `LOGICAL*2`	`Cshort`	`Int16`
`unsigned short`		`Cushort`	`UInt16`
`int`, `BOOL` (C, typical)	`INTEGER*4`, `LOGICAL*4`	`Cint`	`Int32`
`unsigned int`		`Cuint`	`UInt32`
`long long`	`INTEGER*8`, `LOGICAL*8`	`Clonglong`	`Int64`
`unsigned long long`		`Culonglong`	`UInt64`
`intmax_t`		`Cintmax_t`	`Int64`
`uintmax_t`		`Cuintmax_t`	`UInt64`
`float`	`REAL*4`	`Cfloat`	`Float32`
`double`	`REAL*8`	`Cdouble`	`Float64`
`complex float`	`COMPLEX*8`	`ComplexF32`	`Complex{Float32}`
`complex double`	`COMPLEX*16`	`ComplexF64`	`Complex{Float64}`
`ptrdiff_t`		`Cptrdiff_t`	`Int`
`ssize_t`		`Cssize_t`	`Int`
`size_t`		`Csize_t`	`UInt`
`void`			`Cvoid`
`void` and `[[noreturn]]` or `_Noreturn`			`Union{}`
`void*`			`Ptr{Cvoid}` (or similarly `Ref{Cvoid}`)
`T*` (where T represents an appropriately defined type)			`Ref{T}` (T may be safely mutated only if T is an isbits type)
`char*` (or `char[]`, e.g. a string)	`CHARACTER*N`		`Cstring` if null-terminated, or `Ptr{UInt8}` if not
`char**` (or `*char[]`)			`Ptr{Ptr{UInt8}}`
`jl_value_t*` (any Julia Type)			`Any`
`jl_value_t* const*` (a reference to a Julia value)			`Ref{Any}` (const, since mutation would require a write barrier, which is not possible to insert correctly)
`va_arg`			Not supported
`...` (variadic function specification)			`T...` (where T is one of the above types, when using the `ccall` function)
`...` (variadic function specification)			`; va_arg1::T, va_arg2::S, etc.` (only supported with `@ccall` macro)

C name	Standard Julia Alias	Julia Base Type
`char`	`Cchar`	`Int8` (x86, x86_64), `UInt8` (powerpc, arm)
`long`	`Clong`	`Int` (UNIX), `Int32` (Windows)
`unsigned long`	`Culong`	`UInt` (UNIX), `UInt32` (Windows)
`wchar_t`	`Cwchar_t`	`Int32` (UNIX), `UInt16` (Windows)

@ccall	ccall
`@ccall clock()::Int32`	`ccall(:clock, Int32, ())`
`@ccall f(a::Cint)::Cint`	`ccall(:f, Cint, (Cint,), a)`
`@ccall "mylib".f(a::Cint, b::Cdouble)::Cvoid`	`ccall((:f, "mylib"), Cvoid, (Cint, Cdouble), a, b)`
`@ccall $fptr.f()::Cvoid`	`ccall(fptr, f, Cvoid, ())`
`@ccall printf("%s = %d\n"::Cstring ; "foo"::Cstring, foo::Cint)::Cint`	`<unavailable>`
`@ccall printf("%s = %s\n"::Cstring ; "2 + 2"::Cstring, "5"::Cstring)::Cint`	`ccall(:printf, Cint, (Cstring, Cstring...), "%s = %s\n", "2 + 2", "5")`
`<unavailable>`	`ccall(:gethostname, stdcall, Int32, (Ptr{UInt8}, UInt32), hn, length(hn))`

Chapter 29

Handling Operating System Variation

When writing cross-platform applications or libraries, it is often necessary to allow for differences between operating systems. The variable `Sys.KERNEL` can be used to handle such cases. There are several functions in the `Sys` module intended to make this easier, such as `isunix`, `islinux`, `isapple`, `isbsd`, `isfreebsd`, and `iswindows`. These may be used as follows:

```
if Sys.iswindows()
    windows_specific_thing(a)
end
```

Note that `islinux`, `isapple`, and `isfreebsd` are mutually exclusive subsets of `isunix`. Additionally, there is a macro `@static` which makes it possible to use these functions to conditionally hide invalid code, as demonstrated in the following examples.

Simple blocks:

```
ccall((@static Sys.iswindows() ? :_fopen : :fopen), ...)
```

Complex blocks:

```
@static if Sys.islinux()
    linux_specific_thing(a)
elseif Sys.isapple()
    apple_specific_thing(a)
else
    generic_thing(a)
end
```

When nesting conditionals, the `@static` must be repeated for each level (parentheses optional, but recommended for readability):

```
@static Sys.iswindows() ? :a : (@static Sys.isapple() ? :b : :c)
```

Chapter 30

Environment Variables

Julia can be configured with a number of environment variables, set either in the usual way for each operating system, or in a portable way from within Julia. Supposing that you want to set the environment variable JULIA_EDITOR to vim, you can type ENV["JULIA_EDITOR"] = "vim" (for instance, in the REPL) to make this change on a case by case basis, or add the same to the user configuration file ~/.julia/config/startup.jl in the user's home directory to have a permanent effect. The current value of the same environment variable can be determined by evaluating ENV["JULIA_EDITOR"].

The environment variables that Julia uses generally start with JULIA. If InteractiveUtils.versioninfo is called with the keyword verbose=true, then the output will list any defined environment variables relevant for Julia, including those which include JULIA in their names.

Note

It is recommended to avoid changing environment variables during runtime, such as within a ~/.julia/config/startup.jl.

One reason is that some julia language variables, such as JULIA_NUM_THREADS and JULIA_PROJECT, need to be set before Julia starts.

Similarly, __init__() functions of user modules in the sysimage (via PackageCompiler) are run before startup.jl, so setting environment variables in a startup.jl may be too late for user code.

Further, changing environment variables during runtime can introduce data races into otherwise benign code.

In Bash, environment variables can either be set manually by running, e.g., export JULIA_NUM_THREADS=4 before starting Julia, or by adding the same command to ~/.bashrc or ~/.bash_profile to set the variable each time Bash is started.

30.1 File locations

JULIA_BINDIR

The absolute path of the directory containing the Julia executable, which sets the global variable Sys.BINDIR. If $JULIA_BINDIR is not set, then Julia determines the value Sys.BINDIR at run-time.

The executable itself is one of

```
$JULIA_BINDIR/julia
$JULIA_BINDIR/julia-debug
```

by default.

The global variable `Base.DATAROOTDIR` determines a relative path from `Sys.BINDIR` to the data directory associated with Julia. Then the path

```
$JULIA_BINDIR/$DATAROOTDIR/julia/base
```

determines the directory in which Julia initially searches for source files (via `Base.find_source_file()`).

Likewise, the global variable `Base.SYSCONFDIR` determines a relative path to the configuration file directory. Then Julia searches for a `startup.jl` file at

```
$JULIA_BINDIR/$SYSCONFDIR/julia/startup.jl
$JULIA_BINDIR/../etc/julia/startup.jl
```

by default (via `Base.load_julia_startup()`).

For example, a Linux installation with a Julia executable located at `/bin/julia`, a `DATAROOTDIR` of `../share`, and a `SYSCONFDIR` of `../etc` will have `JULIA_BINDIR` set to `/bin`, a source-file search path of

```
/share/julia/base
```

and a global configuration search path of

```
/etc/julia/startup.jl
```

JULIA_PROJECT

A directory path that indicates which project should be the initial active project. Setting this environment variable has the same effect as specifying the `--project` start-up option, but `--project` has higher precedence. If the variable is set to `@.` (note the trailing dot) then Julia tries to find a project directory that contains `Project.toml` or `JuliaProject.toml` file from the current directory and its parents. See also the chapter on Code Loading.

Note

`JULIA_PROJECT` must be defined before starting julia; defining it in `startup.jl` is too late in the startup process.

JULIA_LOAD_PATH

The `JULIA_LOAD_PATH` environment variable is used to populate the global Julia `LOAD_PATH` variable, which determines which packages can be loaded via `import` and `using` (see Code Loading).

Unlike the shell `PATH` variable, empty entries in `JULIA_LOAD_PATH` are expanded to the default value of `LOAD_PATH`, `["@", "@v#.#", "@stdlib"]` when populating `LOAD_PATH`. This allows easy appending, prepending, etc. of the load path value in shell scripts regardless of whether `JULIA_LOAD_PATH` is already set or not. For example, to prepend the directory `/foo/bar` to `LOAD_PATH` just do

```
export JULIA_LOAD_PATH="/foo/bar:$JULIA_LOAD_PATH"
```

If the JULIA_LOAD_PATH environment variable is already set, its old value will be prepended with `/foo/bar`. On the other hand, if JULIA_LOAD_PATH is not set, then it will be set to `/foo/bar:` which will expand to a LOAD_PATH value of `["/foo/bar", "@", "@v#.#", "@stdlib"]`. If JULIA_LOAD_PATH is set to the empty string, it expands to an empty LOAD_PATH array. In other words, the empty string is interpreted as a zero-element array, not a one-element array of the empty string. This behavior was chosen so that it would be possible to set an empty load path via the environment variable. If you want the default load path, either unset the environment variable or if it must have a value, set it to the string `:`.

Note

On Windows, path elements are separated by the `;` character, as is the case with most path lists on Windows. Replace `:` with `;` in the above paragraph.

JULIA_DEPOT_PATH

The JULIA_DEPOT_PATH environment variable is used to populate the global Julia DEPOT_PATH variable, which controls where the package manager, as well as Julia's code loading mechanisms, look for package registries, installed packages, named environments, repo clones, cached compiled package images, configuration files, and the default location of the REPL's history file.

Unlike the shell PATH variable but similar to JULIA_LOAD_PATH, empty entries in JULIA_DEPOT_PATH have special behavior:

- At the end, it is expanded to the default value of DEPOT_PATH, *excluding* the user depot.
- At the start, it is expanded to the default value of DEPOT_PATH, *including* the user depot.

This allows easy overriding of the user depot, while still retaining access to resources that are bundled with Julia, like cache files, artifacts, etc. For example, to switch the user depot to `/foo/bar` use a trailing `:`

```
export JULIA_DEPOT_PATH="/foo/bar:"
```

All package operations, like cloning registries or installing packages, will now write to `/foo/bar`, but since the empty entry is expanded to the default system depot, any bundled resources will still be available. If you really only want to use the depot at `/foo/bar`, and not load any bundled resources, simply set the environment variable to `/foo/bar` without the trailing colon.

To append a depot at the end of the full default list, including the default user depot, use a leading `:`

```
export JULIA_DEPOT_PATH=":/foo/bar"
```

There are two exceptions to the above rule. First, if JULIA_DEPOT_PATH is set to the empty string, it expands to an empty DEPOT_PATH array. In other words, the empty string is interpreted as a zero-element array, not a one-element array of the empty string. This behavior was chosen so that it would be possible to set an empty depot path via the environment variable.

Second, if no user depot is specified in JULIA_DEPOT_PATH, then the empty entry is expanded to the default depot *including* the user depot. This makes it possible to use the default depot, as if the environment variable was unset, by setting it to the string `:`.

Note

On Windows, path elements are separated by the `;` character, as is the case with most path lists on Windows. Replace `:` with `;` in the above paragraph.

Note

JULIA_DEPOT_PATH must be defined before starting julia; defining it in `startup.jl` is too late in the startup process; at that point you can instead directly modify the DEPOT_PATH array, which is populated from the environment variable.

JULIA_HISTORY

The absolute path `REPL.find_hist_file()` of the REPL's history file. If `$JULIA_HISTORY` is not set, then `REPL.find_hist_file()` defaults to

```
$(DEPOT_PATH[1])/logs/repl_history.jl
```

JULIA_MAX_NUM_PRECOMPILE_FILES

Sets the maximum number of different instances of a single package that are to be stored in the precompile cache (default = 10).

JULIA_VERBOSE_LINKING

If set to true, linker commands will be displayed during precompilation.

30.2 Pkg.jl

JULIA_CI

If set to `true`, this indicates to the package server that any package operations are part of a continuous integration (CI) system for the purposes of gathering package usage statistics.

JULIA_NUM_PRECOMPILE_TASKS

The number of parallel tasks to use when precompiling packages. See `Pkg.precompile`.

JULIA_PKG_DEVDIR

The default directory used by `Pkg.develop` for downloading packages.

JULIA_PKG_IGNORE_HASHES

If set to 1, this will ignore incorrect hashes in artifacts. This should be used carefully, as it disables verification of downloads, but can resolve issues when moving files across different types of file systems. See Pkg.jl issue #2317 for more details.

Julia 1.6

This is only supported in Julia 1.6 and above.

JULIA_PKG_OFFLINE

If set to `true`, this will enable offline mode: see `Pkg.offline`.

Julia 1.5

Pkg's offline mode requires Julia 1.5 or later.

JULIA_PKG_PRECOMPILE_AUTO

If set to `0`, this will disable automatic precompilation by package actions which change the manifest. See `Pkg.precompile`.

JULIA_PKG_SERVER

Specifies the URL of the package registry to use. By default, Pkg uses `https://pkg.julialang.org` to fetch Julia packages. In addition, you can disable the use of the PkgServer protocol, and instead access the packages directly from their hosts (GitHub, GitLab, etc.) by setting: `export JULIA_PKG_SERVER=""`

JULIA_PKG_SERVER_REGISTRY_PREFERENCE

Specifies the preferred registry flavor. Currently supported values are `conservative` (the default), which will only publish resources that have been processed by the storage server (and thereby have a higher probability of being available from the PkgServers), whereas `eager` will publish registries whose resources have not necessarily been processed by the storage servers. Users behind restrictive firewalls that do not allow downloading from arbitrary servers should not use the `eager` flavor.

Julia 1.7

This only affects Julia 1.7 and above.

JULIA_PKG_UNPACK_REGISTRY

If set to `true`, this will unpack the registry instead of storing it as a compressed tarball.

Julia 1.7

This only affects Julia 1.7 and above. Earlier versions will always unpack the registry.

JULIA_PKG_USE_CLI_GIT

If set to `true`, Pkg operations which use the git protocol will use an external `git` executable instead of the default libgit2 library.

Julia 1.7

Use of the `git` executable is only supported on Julia 1.7 and above.

JULIA_PKGRESOLVE_ACCURACY

The accuracy of the package resolver. This should be a positive integer, the default is 1.

JULIA_PKG_PRESERVE_TIERED_INSTALLED

Change the default package installation strategy to `Pkg.PRESERVE_TIERED_INSTALLED` to let the package manager try to install versions of packages while keeping as many versions of packages already installed as possible.

Julia 1.9

This only affects Julia 1.9 and above.

JULIA_PKG_GC_AUTO

If set to `false`, automatic garbage collection of packages and artifacts will be disabled; see `Pkg.gc` for more details.

Julia 1.12

This environment variable is only supported on Julia 1.12 and above.

30.3 Network transport

JULIA_NO_VERIFY_HOSTS

JULIA_SSL_NO_VERIFY_HOSTS

JULIA_SSH_NO_VERIFY_HOSTS

JULIA_ALWAYS_VERIFY_HOSTS

Specify hosts whose identity should or should not be verified for specific transport layers. See `NetworkOptions.verify_host`

JULIA_SSL_CA_ROOTS_PATH

Specify the file or directory containing the certificate authority roots. See `NetworkOptions.ca_roots`

30.4 External applications

JULIA_SHELL

The absolute path of the shell with which Julia should execute external commands (via `Base.repl_cmd()`). Defaults to the environment variable `$SHELL`, and falls back to `/bin/sh` if `$SHELL` is unset.

Note

On Windows, this environment variable is ignored, and external commands are executed directly.

JULIA_EDITOR

The editor returned by `InteractiveUtils.editor()` and used in, e.g., `InteractiveUtils.edit`, referring to the command of the preferred editor, for instance `vim`.

`$JULIA_EDITOR` takes precedence over `$VISUAL`, which in turn takes precedence over `$EDITOR`. If none of these environment variables is set, then the editor is taken to be open on Windows and OS X, or `/etc/alternatives/editor` if it exists, or `emacs` otherwise.

To use Visual Studio Code on Windows, set `$JULIA_EDITOR` to `code.cmd`.

30.5 Parallelization

JULIA_CPU_THREADS

Overrides the global variable `Base.Sys.CPU_THREADS`, the number of logical CPU cores available.

JULIA_WORKER_TIMEOUT

A `Float64` that sets the value of `Distributed.worker_timeout()` (default: `60.0`). This function gives the number of seconds a worker process will wait for a master process to establish a connection before dying.

JULIA_NUM_THREADS

An unsigned 64-bit integer (`uint64_t`) or string that sets the maximum number of threads available to Julia. If `$JULIA_NUM_THREADS` is not set or is a non-positive integer, or if the number of CPU threads cannot be determined through system calls, then the number of threads is set to 1.

If `$JULIA_NUM_THREADS` is set to `auto`, then the number of threads will be set to the number of CPU threads. It can also be set to a comma-separated string to specify the size of the `:default` and `:interactive` threadpools, respectively:

```
# 5 threads in the :default pool and 2 in the :interactive pool
export JULIA_NUM_THREADS=5,2

# `auto` threads in the :default pool and 1 in the :interactive pool
export JULIA_NUM_THREADS=auto,1
```

Note

JULIA_NUM_THREADS must be defined before starting Julia; defining it in `startup.jl` is too late in the startup process.

Julia 1.5

In Julia 1.5 and above the number of threads can also be specified on startup using the `-t`/`--threads` command line argument.

Julia 1.7

The `auto` value for `$JULIA_NUM_THREADS` requires Julia 1.7 or above.

Julia 1.9

The `x,y` format for threadpools requires Julia 1.9 or above.

JULIA_THREAD_SLEEP_THRESHOLD

If set to a string that starts with the case-insensitive substring `"infinite"`, then spinning threads never sleep. Otherwise, `$JULIA_THREAD_SLEEP_THRESHOLD` is interpreted as an unsigned 64-bit integer (`uint64_t`) and gives, in nanoseconds, the amount of time after which spinning threads should sleep.

JULIA_NUM_GC_THREADS

Sets the number of threads used by Garbage Collection. If unspecified is set to the number of worker threads.

Julia 1.10

The environment variable was added in 1.10

JULIA_IMAGE_THREADS

An unsigned 32-bit integer that sets the number of threads used by image compilation in this Julia process. The value of this variable may be ignored if the module is a small module. If left unspecified, the smaller of the value of JULIA_CPU_THREADS or half the number of logical CPU cores is used in its place.

JULIA_IMAGE_TIMINGS

A boolean value that determines if detailed timing information is printed during during image compilation. Defaults to 0.

JULIA_EXCLUSIVE

If set to anything besides `0`, then Julia's thread policy is consistent with running on a dedicated machine: each thread in the default threadpool is affinitized. Interactive threads remain under the control of the operating system scheduler.

Otherwise, Julia lets the operating system handle thread policy.

30.6 Garbage Collection

JULIA_HEAP_SIZE_HINT

Environment variable equivalent to the `--heap-size-hint=<size>[<unit>]` command line option.

Forces garbage collection if memory usage is higher than the given value. The value may be specified as a number of bytes, optionally in units of:

```
- B  (bytes)
- K  (kibibytes)
- M  (mebibytes)
- G  (gibibytes)
- T  (tebibytes)
- %  (percentage of physical memory)
```

For example, `JULIA_HEAP_SIZE_HINT=1G` would provide a 1 GB heap size hint to the garbage collector.

30.7 REPL formatting

Environment variables that determine how REPL output should be formatted at the terminal. The `JULIA_*_COLOR` variables should be set to ANSI terminal escape sequences. Julia provides a high-level interface with much of the same functionality; see the section on The Julia REPL.

JULIA_ERROR_COLOR

The formatting `Base.error_color()` (default: light red, `"\033[91m"`) that errors should have at the terminal.

JULIA_WARN_COLOR

The formatting `Base.warn_color()` (default: yellow, `"\033[93m"`) that warnings should have at the terminal.

JULIA_INFO_COLOR

The formatting `Base.info_color()` (default: cyan, `"\033[36m"`) that info should have at the terminal.

JULIA_INPUT_COLOR

The formatting `Base.input_color()` (default: normal, `"\033[0m"`) that input should have at the terminal.

JULIA_ANSWER_COLOR

The formatting `Base.answer_color()` (default: normal, `"\033[0m"`) that output should have at the terminal.

NO_COLOR

When this variable is present and not an empty string (regardless of its value) then colored text will be disabled on the REPL. Can be overridden with the flag `--color=yes` or with the environment variable FORCE_COLOR. This environment variable is commonly recognized by command-line applications.

FORCE_COLOR

When this variable is present and not an empty string (regardless of its value) then colored text will be enabled on the REPL. Can be overridden with the flag `--color=no`. This environment variable is commonly recognized by command-line applications.

30.8 System and Package Image Building

JULIA_CPU_TARGET

Modify the target machine architecture for (pre)compiling system and package images. JULIA_CPU_TARGET only affects machine code image generation being output to a disk cache. Unlike the `--cpu-target`, or `-C`, command line option, it does not influence just-in-time (JIT) code generation within a Julia session where machine code is only stored in memory.

Valid values for JULIA_CPU_TARGET can be obtained by executing `julia -C help`.

To get the CPU target string that was used to build the current system image, use `Sys.sysimage_target()`. This can be useful for reproducing the same system image or understanding what CPU features were enabled during compilation.

Setting JULIA_CPU_TARGET is important for heterogeneous compute systems where processors of distinct types or features may be present. This is commonly encountered in high performance computing (HPC) clusters since the component nodes may be using distinct processors. In this case, you may want to use the `sysimage` CPU target to maintain the same configuration as the sysimage. See below for more details.

The CPU target string is a list of strings separated by `;` each string starts with a CPU or architecture name and followed by an optional list of features separated by `,`. A `generic` or empty CPU name means the basic required feature set of the target ISA which is at least the architecture the C/C++ runtime is compiled with. Each string is interpreted by LLVM.

Note

Package images can only target the same or more specific CPU features than their base system image.

A few special features are supported:

1. `sysimage`

   A special keyword that can be used as a CPU target name, which will be replaced with the CPU target string that was used to build the current system image. This allows you to specify CPU targets that build upon or extend the current sysimage's target, which is particularly helpful for creating package images that are as flexible as the sysimage.

2. `clone_all`

   This forces the target to have all functions in sysimg cloned. When used in negative form (i.e. `-clone_all`), this disables full clone that's enabled by default for certain targets.

3. `base([0-9]*)`

   This specifies the (0-based) base target index. The base target is the target that the current target is based on, i.e. the functions that are not being cloned will use the version in the base target. This option causes the base target to be fully cloned (as if `clone_all` is specified for it) if it is not the default target (0). The index can only be smaller than the current index.

4. `opt_size`

   Optimize for size with minimum performance impact. Clang/GCC's `-Os`.

5. `min_size`

   Optimize only for size. Clang's `-Oz`.

30.9 Debugging and profiling

JULIA_DEBUG

Enable debug logging for a file or module, see `Logging` for more information.

CI Debug Environment Variables

Julia automatically enables verbose debugging options when certain continuous integration (CI) debug environment variables are set. This improves the debugging experience when CI jobs are re-run with debug logging enabled, by automatically:

- Enabling `--trace-eval` (location mode) to show expressions being evaluated
- Setting `JULIA_TEST_VERBOSE=true` to enable verbose test output

This allows developers to get detailed debugging information from CI runs without modifying their scripts or workflow files.

JULIA_PROFILE_PEEK_HEAP_SNAPSHOT

Enable collecting of a heap snapshot during execution via the profiling peek mechanism. See Triggered During Execution.

JULIA_TIMING_SUBSYSTEMS

Allows you to enable or disable zones for a specific Julia run. For instance, setting the variable to `+GC,-INFERENCE` will enable the `GC` zones and disable the `INFERENCE` zones. See Dynamically Enabling and Disabling Zones.

JULIA_GC_WAIT_FOR_DEBUGGER

If set to anything besides `0`, then the Julia garbage collector will wait for a debugger to attach instead of aborting whenever there's a critical error.

Note

This environment variable only has an effect if Julia was compiled with garbage-collection debugging (that is, if `WITH_GC_DEBUG_ENV` is set to 1 in the build configuration).

ENABLE_JITPROFILING

If set to anything besides `0`, then the compiler will create and register an event listener for just-in-time (JIT) profiling.

Note

This environment variable only has an effect if Julia was compiled with JIT profiling support, using either

- Intel's VTune™ Amplifier (`USE_INTEL_JITEVENTS` set to 1 in the build configuration), or
- OProfile (`USE_OPROFILE_JITEVENTS` set to 1 in the build configuration).
- Perf (`USE_PERF_JITEVENTS` set to 1 in the build configuration). This integration is enabled by default.

ENABLE_GDBLISTENER

If set to anything besides `0` enables GDB registration of Julia code on release builds. On debug builds of Julia this is always enabled. Recommended to use with `-g 2`.

JULIA_LLVM_ARGS

Arguments to be passed to the LLVM backend.

JULIA_FALLBACK_REPL

Forces the fallback repl instead of REPL.jl.

JULIA_LOAD_CODEGEN_LIB

If set to a false value (`0`, `f`, `false`, `n`, or `no`, case-insensitive), the loader does not load `libjulia-codegen`, and the fallback (interpreter-only) implementations in `libjulia-internal` are used instead — exactly as if the library were absent from the installation. Intended for testing and debugging the no-codegen configuration.

Chapter 31

Embedding Julia

As we have seen in Calling C and Fortran Code, Julia has a simple and efficient way to call functions written in C. But there are situations where the opposite is needed: calling Julia functions from C code. This can be used to integrate Julia code into a larger C/C++ project, without the need to rewrite everything in C/C++. Julia has a C API to make this possible. As almost all programming languages have some way to call C functions, the Julia C API can also be used to build further language bridges (e.g. calling Julia from Python, Rust or C#). Even though Rust and C++ can use the C embedding API directly, both have packages helping with it, for C++ Jluna is useful.

31.1 High-Level Embedding

Note: This section covers embedding Julia code in C on Unix-like operating systems. For doing this on Windows, please see the section following this, High-Level Embedding on Windows with Visual Studio.

We start with a simple C program that initializes Julia and calls some Julia code:

```
#include <julia.h>
JULIA_DEFINE_FAST_TLS // only define this once, in an executable (not in a shared library) if you
↪  want fast code.

int main(int argc, char *argv[])
{
    /* required: setup the Julia context */
    jl_init();

    /* run Julia commands */
    jl_eval_string("print(sqrt(2.0))");

    /* strongly recommended: notify Julia that the
         program is about to terminate. this allows
         Julia time to cleanup pending write requests
         and run all finalizers
    */
    jl_atexit_hook(0);
    return 0;
}
```

In order to build this program you must add the path to the Julia header to the include path and link against `libjulia`. For instance, when Julia is installed to `$JULIA_DIR`, one can compile the above test program `test.c` with `gcc` using:

```
gcc -o test -fPIC -I$JULIA_DIR/include/julia -L$JULIA_DIR/lib -Wl,-rpath,$JULIA_DIR/lib test.c
↪  -ljulia
```

Alternatively, look at the `embedding.c` program in the Julia source tree in the `test/embedding/` folder. The file `cli/loader_exe.c` program is another simple example of how to set `jl_options` options while linking against `libjulia`.

The first thing that must be done before calling any other Julia C function is to initialize Julia. This is done by calling `jl_init`, which tries to automatically determine Julia's install location. If you need to specify a custom location, or specify which system image to load, use `jl_init_with_image_file` or `jl_init_with_image_handle` instead.

The second statement in the test program evaluates a Julia statement using a call to `jl_eval_string`.

Before the program terminates, it is strongly recommended that `jl_atexit_hook` is called. The above example program calls this just before returning from `main`.

Note

Currently, dynamically linking with the `libjulia` shared library requires passing the RTLD_GLOBAL option. In Python, this looks like:

```
>>> julia=CDLL('./libjulia.dylib',RTLD_GLOBAL)
>>> julia.jl_init.argtypes = []
>>> julia.jl_init()
250593296
```

Note

If the julia program needs to access symbols from the main executable, it may be necessary to add the `-Wl,--export-dynamic` linker flag at compile time on Linux in addition to the ones generated by `julia-config.jl` described below. This is not necessary when compiling a shared library.

Using julia-config to automatically determine build parameters

The script `julia-config.jl` was created to aid in determining what build parameters are required by a program that uses embedded Julia. This script uses the build parameters and system configuration of the particular Julia distribution it is invoked by to export the necessary compiler flags for an embedding program to interact with that distribution. This script is located in the Julia shared data directory.

Example

```
#include <julia.h>

int main(int argc, char *argv[])
{
    jl_init();
```

```
    (void)jl_eval_string("println(sqrt(2.0))");
    jl_atexit_hook(0);
    return 0;
}
```

On the command line

A simple use of this script is from the command line. Assuming that `julia-config.jl` is located in `/usr/local/julia/share/julia`, it can be invoked on the command line directly and takes any combination of three flags:

```
/usr/local/julia/share/julia/julia-config.jl
Usage: julia-config [--cflags|--ldflags|--ldlibs]
```

If the above example source is saved in the file `embed_example.c`, then the following command will compile it into an executable program on Linux and Windows (MSYS2 environment). On macOS, substitute `clang` for `gcc`.:

```
/usr/local/julia/share/julia/julia-config.jl --cflags --ldflags --ldlibs | xargs gcc
↪  embed_example.c
```

Use in Makefiles

In general, embedding projects will be more complicated than the above example, and so the following allows general makefile support as well – assuming GNU make because of the use of the **shell** macro expansions. Furthermore, although `julia-config.jl` is usually in the `/usr/local` directory, if it isn't, then Julia itself can be used to find `julia-config.jl`, and the makefile can take advantage of this. The above example is extended to use a makefile:

```
JL_SHARE = $(shell julia -e 'print(joinpath(Sys.BINDIR, Base.DATAROOTDIR, "julia"))')
CFLAGS   += $(shell $(JL_SHARE)/julia-config.jl --cflags)
CXXFLAGS += $(shell $(JL_SHARE)/julia-config.jl --cflags)
LDFLAGS  += $(shell $(JL_SHARE)/julia-config.jl --ldflags)
LDLIBS   += $(shell $(JL_SHARE)/julia-config.jl --ldlibs)

all: embed_example
```

Now the build command is simply `make`.

31.2 High-Level Embedding on Windows with Visual Studio

If the `JULIA_DIR` environment variable hasn't been setup, add it using the System panel before starting Visual Studio. The `bin` folder under JULIA_DIR should be on the system PATH.

We start by opening Visual Studio and creating a new Console Application project. Open the 'stdafx.h' header file, and add the following lines at the end:

```
#include <julia.h>
```

Then, replace the main() function in the project with this code:

```
int main(int argc, char *argv[])
{
    /* required: setup the Julia context */
    jl_init();

    /* run Julia commands */
    jl_eval_string("print(sqrt(2.0))");

    /* strongly recommended: notify Julia that the
         program is about to terminate. this allows
         Julia time to cleanup pending write requests
         and run all finalizers
    */
    jl_atexit_hook(0);
    return 0;
}
```

The next step is to set up the project to find the Julia include files and the libraries. It's important to know whether the Julia installation is 32- or 64-bit. Remove any platform configuration that doesn't correspond to the Julia installation before proceeding.

Using the project Properties dialog, go to `C/C++` | `General` and add `$(JULIA_DIR)\include\julia\` to the Additional Include Directories property. Then, go to the `Linker` | `General` section and add `$(JULIA_DIR)\lib` to the Additional Library Directories property. Finally, under `Linker` | `Input`, add `libjulia.dll.a;libopenlibm.dll.a;` to the list of libraries.

At this point, the project should build and run.

31.3 Converting Types

Real applications will not only need to execute expressions, but also return their values to the host program. `jl_eval_string` returns a `jl_value_t*`, which is a pointer to a heap-allocated Julia object. Storing simple data types like `Float64` in this way is called `boxing`, and extracting the stored primitive data is called `unboxing`. Our improved sample program that calculates the square root of 2 in Julia and reads back the result in C has a body that now contains this code:

```
jl_value_t *ret = jl_eval_string("sqrt(2.0)");

if (jl_typeis(ret, jl_float64_type)) {
    double ret_unboxed = jl_unbox_float64(ret);
    printf("sqrt(2.0) in C: %e \n", ret_unboxed);
}
else {
    printf("ERROR: unexpected return type from sqrt(::Float64)\n");
}
```

In order to check whether `ret` is of a specific Julia type, we can use the `jl_isa`, `jl_typeis`, or `jl_is_...` functions. By typing `typeof(sqrt(2.0))` into the Julia shell we can see that the return type is `Float64` (`double` in C). To convert the boxed Julia value into a C double the `jl_unbox_float64` function is used in the above code snippet.

Corresponding `jl_box_...` functions are used to convert the other way:

```
jl_value_t *a = jl_box_float64(3.0);
jl_value_t *b = jl_box_float32(3.0f);
jl_value_t *c = jl_box_int32(3);
```

As we will see next, boxing is required to call Julia functions with specific arguments.

31.4 Calling Julia Functions

While `jl_eval_string` allows C to obtain the result of a Julia expression, it does not allow passing arguments computed in C to Julia. For this you will need to invoke Julia functions directly, using `jl_call`:

```
jl_value_t *func = jl_get_function(jl_base_module, "sqrt");
jl_value_t *argument = jl_box_float64(2.0);
jl_value_t *ret = jl_call1(func, argument);
```

In the first step, a handle to the Julia function `sqrt` is retrieved by calling `jl_get_function`. The first argument passed to `jl_get_function` is a pointer to the `Base` module in which `sqrt` is defined. Then, the double value is boxed using `jl_box_float64`. Finally, in the last step, the function is called using `jl_call1`. `jl_call0`, `jl_call2`, and `jl_call3` functions also exist, to conveniently handle different numbers of arguments. To pass more arguments, use `jl_call`:

```
jl_value_t *jl_call(jl_value_t *f, jl_value_t **args, int32_t nargs)
```

Its second argument `args` is an array of `jl_value_t*` arguments and `nargs` is the number of arguments.

There is also an alternative, possibly simpler, way of calling Julia functions and that is via `@cfunction`. Using `@cfunction` allows you to do the type conversions on the Julia side, which is typically easier than doing it on the C side. The `sqrt` example above would with `@cfunction` be written as:

```
double (*sqrt_jl)(double) = jl_unbox_voidpointer(jl_eval_string("@cfunction(sqrt, Float64,
↪  (Float64,))"));
double ret = sqrt_jl(2.0);
```

where we first define a C callable function in Julia, extract the function pointer from it, and finally call it. In addition to simplifying type conversions by doing them in the higher-level language, calling Julia functions via `@cfunction` pointers eliminates the dynamic-dispatch overhead required by `jl_call` (for which all of the arguments are "boxed"), and should have performance equivalent to native C function pointers.

31.5 Memory Management

As we have seen, Julia objects are represented in C as pointers of type `jl_value_t*`. This raises the question of who is responsible for freeing these objects.

Typically, Julia objects are freed by the garbage collector (GC), but the GC does not automatically know that we are holding a reference to a Julia value from C. This means the GC can free objects out from under you, rendering pointers invalid.

The GC will only run when new Julia objects are being allocated. Calls like `jl_box_float64` perform allocation, but allocation might also happen at any point in running Julia code.

When writing code that embeds Julia, it is generally safe to use `jl_value_t*` values in between `jl_...` calls (as GC will only get triggered by those calls). But in order to make sure that values can survive `jl_...` calls, we have to tell Julia that we still hold a reference to Julia root values, a process called "GC rooting". Rooting a value will ensure that the garbage collector does not accidentally identify this value as unused and free the memory backing that value. This can be done using the `JL_GC_PUSH` macros:

```
jl_value_t *ret = jl_eval_string("sqrt(2.0)");
JL_GC_PUSH1(&ret);
// Do something with ret
JL_GC_POP();
```

The `JL_GC_POP` call releases the references established by the previous `JL_GC_PUSH`. Note that `JL_GC_PUSH` stores references on the C stack, so it must be exactly paired with a `JL_GC_POP` before the scope is exited. That is, before the function returns, or control flow otherwise leaves the block in which the `JL_GC_PUSH` was invoked.

Several Julia values can be pushed at once using the `JL_GC_PUSH2` to `JL_GC_PUSH6` macros:

```
JL_GC_PUSH2(&ret1, &ret2);
// ...
JL_GC_PUSH6(&ret1, &ret2, &ret3, &ret4, &ret5, &ret6);
```

To push an array of Julia values one can use the `JL_GC_PUSHARGS` macro, which can be used as follows:

```
jl_value_t **args;
JL_GC_PUSHARGS(args, 2); // args can now hold 2 `jl_value_t*` objects
args[0] = some_value;
args[1] = some_other_value;
// Do something with args (e.g. call jl_... functions)
JL_GC_POP();
```

Each scope must have only one call to `JL_GC_PUSH*`, and should be paired with only a single `JL_GC_POP` call. If all necessary variables you want to root cannot be pushed by a one single call to `JL_GC_PUSH*`, or if there are more than 6 variables to be pushed and using an array of arguments is not an option, then one can use inner blocks:

```
jl_value_t *ret1 = jl_eval_string("sqrt(2.0)");
JL_GC_PUSH1(&ret1);
jl_value_t *ret2 = 0;
{
    jl_value_t *func = jl_get_function(jl_base_module, "exp");
    ret2 = jl_call1(func, ret1);
    JL_GC_PUSH1(&ret2);
    // Do something with ret2.
    JL_GC_POP();    // This pops ret2.
}
JL_GC_POP();    // This pops ret1.
```

Note that it is not necessary to have valid `jl_value_t*` values before calling `JL_GC_PUSH*`. It is fine to have a number of them initialized to `NULL`, pass those to `JL_GC_PUSH*` and then create the actual Julia values. For example:

```
jl_value_t *ret1 = NULL, *ret2 = NULL;
JL_GC_PUSH2(&ret1, &ret2);
ret1 = jl_eval_string("sqrt(2.0)");
ret2 = jl_eval_string("sqrt(3.0)");
// Use ret1 and ret2
JL_GC_POP();
```

If it is required to hold the pointer to a variable between functions (or block scopes), then it is not possible to use `JL_GC_PUSH*`. In this case, it is necessary to create and keep a reference to the variable in the Julia global scope. One simple way to accomplish this is to use a global `IdDict` that will hold the references, avoiding deallocation by the GC. However, this method will only work properly with mutable types.

```
// This functions shall be executed only once, during the initialization.
jl_value_t* refs = jl_eval_string("refs = IdDict()");
jl_value_t* setindex = jl_get_function(jl_base_module, "setindex!");

...

// `var` is the variable we want to protect between function calls.
jl_value_t* var = 0;

...

// `var` is a `Vector{Float64}`, which is mutable.
var = jl_eval_string("[sqrt(2.0); sqrt(4.0); sqrt(6.0)]");

// To protect `var`, add its reference to `refs`.
jl_call3(setindex, refs, var, var);
```

If the variable is immutable, then it needs to be wrapped in an equivalent mutable container or, preferably, in a `RefValue{Any}` before it is pushed to `IdDict`. In this approach, the container has to be created or filled in via C code using, for example, the function `jl_new_struct`. If the container is created by `jl_call*`, then you will need to reload the pointer to be used in C code.

```
// This functions shall be executed only once, during the initialization.
jl_value_t* refs = jl_eval_string("refs = IdDict()");
jl_value_t* setindex = jl_get_function(jl_base_module, "setindex!");
jl_datatype_t* reft = (jl_datatype_t*)jl_eval_string("Base.RefValue{Any}");

...

// `var` is the variable we want to protect between function calls.
jl_value_t* var = 0;

...

// `var` is a `Float64`, which is immutable.
var = jl_eval_string("sqrt(2.0)");

// Protect `var` until we add its reference to `refs`.
JL_GC_PUSH1(&var);
```

```
// Wrap `var` in `RefValue{Any}` and push to `refs` to protect it.
jl_value_t* rvar = jl_new_struct(reft, var);
JL_GC_POP();

jl_call3(setindex, refs, rvar, rvar);
```

The GC can be allowed to deallocate a variable by removing the reference to it from `refs` using the function `delete!`, provided that no other reference to the variable is kept anywhere:

```
jl_value_t* delete = jl_get_function(jl_base_module, "delete!");
jl_call2(delete, refs, rvar);
```

As an alternative for very simple cases, it is possible to just create a global container of type `Vector{Any}` and fetch the elements from that when necessary, or even to create one global variable per pointer using

```
jl_module_t *mod = jl_main_module;
jl_sym_t *var = jl_symbol("var");
jl_binding_t *bp = jl_get_binding_wr(mod, var, 1);
jl_checked_assignment(bp, mod, var, val);
```

Updating fields of GC-managed objects

The garbage collector also operates under the assumption that it is aware of every older-generation object pointing to a younger-generation one. Any time a pointer is updated breaking that assumption, it must be signaled to the collector with the `jl_gc_wb` (write barrier) function like so:

```
jl_value_t *parent = some_old_value, *child = some_young_value;
((some_specific_type*)parent)->field = child;
jl_gc_wb(parent, child);
```

It is in general impossible to predict which values will be old at runtime, so the write barrier must be inserted after all explicit stores. One notable exception is if the `parent` object has just been allocated and no garbage collection has run since then. Note that most `jl_...` functions can sometimes invoke garbage collection.

The write barrier is also necessary for arrays of pointers when updating their data directly. Calling `jl_array_ptr_set` is usually much preferred. But direct updates can be done. For example:

```
jl_array_t *some_array = ...; // e.g. a Vector{Any}
void **data = jl_array_data(some_array, void*);
jl_value_t *some_value = ...;
data[0] = some_value;
jl_gc_wb(jl_array_owner(some_array), some_value);
```

Controlling the Garbage Collector

There are some functions to control the GC. In normal use cases, these should not be necessary.

Function	Description
`jl_gc_collect(JL_GC_FULL)`	Force a GC run on all objects
`jl_gc_collect(JL_GC_INCREMENTAL)`	Force a GC run only on young objects
`jl_gc_collect(JL_GC_AUTO)`	Force a GC run, automatically choosing between full and incremental
`jl_gc_enable(0)`	Disable the GC, return previous state as int
`jl_gc_enable(1)`	Enable the GC, return previous state as int
`jl_gc_is_enabled()`	Return current state as int

31.6 Working with Arrays

Julia and C can share array data without copying. The next example will show how this works.

Julia arrays are represented in C by the datatype `jl_array_t*`. Basically, `jl_array_t` is a struct that contains:

- Information about the datatype
- A pointer to the data block
- Information about the sizes of the array

To keep things simple, we start with a 1D array. Creating an array containing Float64 elements of length 10 can be done like this:

```
jl_value_t* array_type = jl_apply_array_type((jl_value_t*)jl_float64_type, 1);
jl_array_t* x          = jl_alloc_array_1d(array_type, 10);
```

Alternatively, if you have already allocated the array you can generate a thin wrapper around its data:

```
double *existingArray = (double*)malloc(sizeof(double)*10);
jl_array_t *x = jl_ptr_to_array_1d(array_type, existingArray, 10, 0);
```

The last argument is a boolean indicating whether Julia should take ownership of the data. If this argument is non-zero, the GC will call `free` on the data pointer when the array is no longer referenced.

In order to access the data of `x`, we can use `jl_array_data`:

```
double *xData = jl_array_data(x, double);
```

Now we can fill the array:

```
for (size_t i = 0; i < jl_array_nrows(x); i++)
    xData[i] = i;
```

Now let us call a Julia function that performs an in-place operation on `x`:

```
jl_value_t *func = jl_get_function(jl_base_module, "reverse!");
jl_call1(func, (jl_value_t*)x);
```

By printing the array, one can verify that the elements of `x` are now reversed.

Accessing Returned Arrays

If a Julia function returns an array, the return value of `jl_eval_string` and `jl_call` can be cast to a `jl_array_t*`:

```
jl_value_t *func  = jl_get_function(jl_base_module, "reverse");
jl_array_t *y = (jl_array_t*)jl_call1(func, (jl_value_t*)x);
```

Now the content of y can be accessed as before using `jl_array_data`. As always, be sure to keep a reference to the array while it is in use.

Multidimensional Arrays

Julia's multidimensional arrays are stored in memory in column-major order. Here is some code that creates a 2D array and accesses its properties:

```
// Create 2D array of float64 type
jl_value_t *array_type = jl_apply_array_type((jl_value_t*)jl_float64_type, 2);
int dims[] = {10,5};
jl_array_t *x  = jl_alloc_array_nd(array_type, dims, 2);

// Get array pointer
double *p = jl_array_data(x, double);
// Get number of dimensions
int ndims = jl_array_ndims(x);
// Get the size of the i-th dim
size_t size0 = jl_array_dim(x,0);
size_t size1 = jl_array_dim(x,1);

// Fill array with data
for(size_t i=0; i<size1; i++)
    for(size_t j=0; j<size0; j++)
        p[j + size0*i] = i + j;
```

Notice that while Julia arrays use 1-based indexing, the C API uses 0-based indexing (for example in calling `jl_array_dim`) in order to read as idiomatic C code.

31.7 Exceptions

Julia code can throw exceptions. For example, consider:

```
jl_eval_string("this_function_does_not_exist()");
```

This call will appear to do nothing. However, it is possible to check whether an exception was thrown:

```
if (jl_exception_occurred())
    printf("%s \n", jl_typeof_str(jl_exception_occurred()));
```

If you are using the Julia C API from a language that supports exceptions (e.g. Python, C#, C++), it makes sense to wrap each call into `libjulia` with a function that checks whether an exception was thrown, and then rethrows the exception in the host language.

Throwing Julia Exceptions

When writing Julia callable functions, it might be necessary to validate arguments and throw exceptions to indicate errors. A typical type check looks like:

```
if (!jl_typeis(val, jl_float64_type)) {
    jl_type_error(function_name, (jl_value_t*)jl_float64_type, val);
}
```

General exceptions can be raised using the functions:

```
void jl_error(const char *str);
void jl_errorf(const char *fmt, ...);
```

`jl_error` takes a C string, and `jl_errorf` is called like `printf`:

```
jl_errorf("argument x = %d is too large", x);
```

where in this example x is assumed to be an integer.

Thread-safety

In general, the Julia C API is not fully thread-safe. When embedding Julia in a multi-threaded application care needs to be taken not to violate the following restrictions:

- `jl_init()` may only be called once in the application life-time. The same applies to `jl_atexit_hook()`, and it may only be called after `jl_init()`.
- `jl_...()` API functions may only be called from the thread in which `jl_init()` was called, *or from threads started by the Julia runtime*. Calling Julia API functions from user-started threads is not supported, and may lead to undefined behaviour and crashes.

The second condition above implies that you can not safely call `jl_...()` functions from threads that were not started by Julia (the thread calling `jl_init()` being the exception). For example, the following is not supported and will most likely segfault:

```
void *func(void*)
{
    // Wrong, jl_eval_string() called from thread that was not started by Julia
    jl_eval_string("println(Threads.threadid())");
    return NULL;
}

int main()
{
    pthread_t t;

    jl_init();
```

```
    // Start a new thread
    pthread_create(&t, NULL, func, NULL);
    pthread_join(t, NULL);

    jl_atexit_hook(0);
}
```

Instead, performing all Julia calls from the same user-created thread will work:

```
void *func(void*)
{
    // Okay, all jl_...() calls from the same thread,
    // even though it is not the main application thread
    jl_init();
    jl_eval_string("println(Threads.threadid())");
    jl_atexit_hook(0);
    return NULL;
}

int main()
{
    pthread_t t;
    // Create a new thread, which runs func()
    pthread_create(&t, NULL, func, NULL);
    pthread_join(t, NULL);
}
```

An example of calling the Julia C API from a thread started by Julia itself:

```
#include <julia/julia.h>
JULIA_DEFINE_FAST_TLS

double c_func(int i)
{
    printf("[C %08x] i = %d\n", pthread_self(), i);

    // Call the Julia sqrt() function to compute the square root of i, and return it
    jl_value_t *sqrt = jl_get_function(jl_base_module, "sqrt");
    jl_value_t* arg = jl_box_int32(i);
    double ret = jl_unbox_float64(jl_call1(sqrt, arg));

    return ret;
}

int main()
{
    jl_init();

    // Define a Julia function func() that calls our c_func() defined in C above
    jl_eval_string("func(i) = ccall(:c_func, Float64, (Int32,), i)");

    // Call func() multiple times, using multiple threads to do so
```

```
    jl_eval_string("println(Threads.threadpoolsize())");
    jl_eval_string("use(i) = println(\"[J $(Threads.threadid())] i = $(i) -> $(func(i))\")");
    jl_eval_string("Threads.@threads for i in 1:5 use(i) end");

    jl_atexit_hook(0);
}
```

If we run this code with 2 Julia threads we get the following output (note: the output will vary per run and system):

```
$ JULIA_NUM_THREADS=2 ./thread_example
2
[C 3bfd9c00] i = 1
[C 23938640] i = 4
[J 1] i = 1 -> 1.0
[C 3bfd9c00] i = 2
[J 1] i = 2 -> 1.4142135623730951
[C 3bfd9c00] i = 3
[J 2] i = 4 -> 2.0
[C 23938640] i = 5
[J 1] i = 3 -> 1.7320508075688772
[J 2] i = 5 -> 2.23606797749979
```

As can be seen, Julia thread 1 corresponds to pthread ID 3bfd9c00, and Julia thread 2 corresponds to ID 23938640, showing that indeed multiple threads are used at the C level, and that we can safely call Julia C API routines from those threads.

Chapter 32

Code Loading

Note

This chapter covers the technical details of package loading. To install packages, use Pkg, Julia's built-in package manager, to add packages to your active environment. To use packages already in your active environment, write `import X` or `using X`, as described in the Modules documentation.

32.1 Definitions

Julia has two mechanisms for loading code:

1. **Code inclusion:** e.g. `include("source.jl")`. Inclusion allows you to split a single program across multiple source files. The expression `include("source.jl")` causes the contents of the file `source.jl` to be evaluated in the global scope of the module where the `include` call occurs. If `include("source.jl")` is called multiple times, `source.jl` is evaluated multiple times. The included path, `source.jl`, is interpreted relative to the file where the `include` call occurs. This makes it simple to relocate a subtree of source files. In the REPL, included paths are interpreted relative to the current working directory, `pwd()`.

2. **Package loading:** e.g. `import X` or `using X`. The import mechanism allows you to load a package—i.e. an independent, reusable collection of Julia code, wrapped in a module—and makes the resulting module available by the name X inside of the importing module. If the same X package is imported multiple times in the same Julia session, it is only loaded the first time—on subsequent imports, the importing module gets a reference to the same module. Note though, that `import X` can load different packages in different contexts: X can refer to one package named X in the main project but potentially to different packages also named X in each dependency. More on this below.

Code inclusion is quite straightforward and simple: it evaluates the given source file in the context of the caller. Package loading is built on top of code inclusion and serves a different purpose. The rest of this chapter focuses on the behavior and mechanics of package loading.

A *package* is a source tree with a standard layout providing functionality that can be reused by other Julia projects. A package is loaded by `import X` or `using X` statements. These statements also make the module named X—which results from loading the package code—available within the module where the import statement occurs. The meaning of X in `import X` is context-dependent: which X package is loaded

depends on what code the statement occurs in. Thus, handling of `import X` happens in two stages: first, it determines **what** package is defined to be X in this context; second, it determines **where** that particular X package is found.

These questions are answered by searching through the project environments listed in `LOAD_PATH` for project files (`Project.toml` or `JuliaProject.toml`), manifest files (`Manifest.toml` or `JuliaManifest.toml`, or the same names suffixed by `-v{major}.{minor}.toml` for specific versions), or folders of source files.

32.2 Federation of packages

Most of the time, a package is uniquely identifiable simply from its name. However, sometimes a project might encounter a situation where it needs to use two different packages that share the same name. While you might be able fix this by renaming one of the packages, being forced to do so can be highly disruptive in a large, shared code base. Instead, Julia's code loading mechanism allows the same package name to refer to different packages in different components of an application.

Julia supports federated package management, which means that multiple independent parties can maintain both public and private packages and registries of packages, and that projects can depend on a mix of public and private packages from different registries. Packages from various registries are installed and managed using a common set of tools and workflows. The Pkg package manager that ships with Julia lets you install and manage your projects' dependencies. It assists in creating and manipulating project files (which describe what other projects that your project depends on), and manifest files (which snapshot exact versions of your project's complete dependency graph).

One consequence of federation is that there cannot be a central authority for package naming. Different entities may use the same name to refer to unrelated packages. This possibility is unavoidable since these entities do not coordinate and may not even know about each other. Because of the lack of a central naming authority, a single project may end up depending on different packages that have the same name. Julia's package loading mechanism does not require package names to be globally unique, even within the dependency graph of a single project. Instead, packages are identified by universally unique identifiers (UUIDs), which get assigned when each package is created. Usually you won't have to work directly with these somewhat cumbersome 128-bit identifiers since Pkg will take care of generating and tracking them for you. However, these UUIDs provide the definitive answer to the question of "*what package does X refer to?*"

Since the decentralized naming problem is somewhat abstract, it may help to walk through a concrete scenario to understand the issue. Suppose you're developing an application called `App`, which uses two packages: `Pub` and `Priv`. `Priv` is a private package that you created, whereas `Pub` is a public package that you use but don't control. When you created `Priv`, there was no public package by the name `Priv`. Subsequently, however, an unrelated package also named `Priv` has been published and become popular. In fact, the `Pub` package has started to use it. Therefore, when you next upgrade `Pub` to get the latest bug fixes and features, `App` will end up depending on two different packages named `Priv`—through no action of yours other than upgrading. `App` has a direct dependency on your private `Priv` package, and an indirect dependency, through `Pub`, on the new public `Priv` package. Since these two `Priv` packages are different but are both required for `App` to continue working correctly, the expression `import Priv` must refer to different `Priv` packages depending on whether it occurs in `App`'s code or in `Pub`'s code. To handle this, Julia's package loading mechanism distinguishes the two `Priv` packages by their UUID and picks the correct one based on its context (the module that called `import`). How this distinction works is determined by environments, as explained in the following sections.

32.3 Environments

An *environment* determines what `import X` and `using X` mean in various code contexts and what files these statements cause to be loaded. Julia understands two kinds of environments:

1. **A project environment** is a directory with a project file and an optional manifest file, and forms an *explicit environment*. The project file determines what the names and identities of the direct dependencies of a project are. The manifest file, if present, gives a complete dependency graph, including all direct and indirect dependencies, exact versions of each dependency, and sufficient information to locate and load the correct version.

2. **A package directory** is a directory containing the source trees of a set of packages as subdirectories, and forms an *implicit environment*. If `X` is a subdirectory of a package directory and `X/src/X.jl` exists, then the package `X` is available in the package directory environment and `X/src/X.jl` is the source file by which it is loaded.

These can be intermixed to create **a stacked environment**: an ordered set of project environments and package directories, overlaid to make a single composite environment. The precedence and visibility rules then combine to determine which packages are available and where they get loaded from. Julia's load path forms a stacked environment, for example.

These environments each serve a different purpose:

- Project environments provide **reproducibility**. By checking a project environment into version control—e.g. a git repository—along with the rest of the project's source code, you can reproduce the exact state of the project and all of its dependencies. The manifest file, in particular, captures the exact version of every dependency, identified by a cryptographic hash of its source tree, which makes it possible for Pkg to retrieve the correct versions and be sure that you are running the exact code that was recorded for all dependencies.

- Package directories provide **convenience** when a full carefully-tracked project environment is unnecessary. They are useful when you want to put a set of packages somewhere and be able to directly use them, without needing to create a project environment for them.

- Stacked environments allow for **adding** tools to the primary environment. You can push an environment of development tools onto the end of the stack to make them available from the REPL and scripts, but not from inside packages.

Note

When loading a package from another environment in the stack other than the active environment the package is loaded in the context of the active environment. This means that the package will be loaded as if it were imported in the active environment, which may affect how its dependencies versions are resolved. When such a package is precompiling it will be marked as a `(serial)` precompile job, which means that its dependencies will be precompiled in series within the same job, which will likely be slower.

At a high-level, each environment conceptually defines three maps: roots, graph and paths. When resolving the meaning of `import X`, the roots and graph maps are used to determine the identity of `X`, while the paths map is used to locate the source code of `X`. The specific roles of the three maps are:

- **roots:** `name::Symbol` ⟶ `uuid::UUID`

  An environment's roots map assigns package names to UUIDs for all the top-level dependencies that the environment makes available to the main project (i.e. the ones that can be loaded in `Main`). When Julia encounters `import X` in the main project, it looks up the identity of X as `roots[:X]`.

- **graph:** `context::UUID` ⟶ `name::Symbol` ⟶ `uuid::UUID`

  An environment's graph is a multilevel map which assigns, for each `context` UUID, a map from names to UUIDs, similar to the roots map but specific to that `context`. When Julia sees `import X` in the code of the package whose UUID is `context`, it looks up the identity of X as `graph[context][:X]`. In particular, this means that `import X` can refer to different packages depending on `context`.

- **paths:** `uuid::UUID` × `name::Symbol` ⟶ `path::String`

  The paths map assigns to each package UUID-name pair, the location of that package's entry-point source file. After the identity of X in `import X` has been resolved to a UUID via roots or graph (depending on whether it is loaded from the main project or a dependency), Julia determines what file to load to acquire X by looking up `paths[uuid,:X]` in the environment. Including this file should define a module named X. Once this package is loaded, any subsequent import resolving to the same `uuid` will create a new binding to the already-loaded package module.

Each kind of environment defines these three maps differently, as detailed in the following sections.

Note

For ease of understanding, the examples throughout this chapter show full data structures for roots, graph and paths. However, Julia's package loading code does not explicitly create these. Instead, it lazily computes only as much of each structure as it needs to load a given package.

Project environments

A project environment is determined by a directory containing a project file called `Project.toml`, and optionally a manifest file called `Manifest.toml`. These files may also be called `JuliaProject.toml` and `JuliaManifest.toml`, in which case `Project.toml` and `Manifest.toml` are ignored. This allows for co-existence with other tools that might consider files called `Project.toml` and `Manifest.toml` significant. For pure Julia projects, however, the names `Project.toml` and `Manifest.toml` are preferred. However, from Julia v1.10.8 onwards, `(Julia)Manifest-v{major}.{minor}.toml` is recognized as a format to make a given julia version use a specific manifest file i.e. in the same folder, a `Manifest-v1.11.toml` would be used by v1.11 and `Manifest.toml` by any other julia version.

The roots, graph and paths maps of a project environment are defined as follows:

The roots map of the environment is determined by the contents of the project file, specifically, its top-level `name` and `uuid` entries and its `[deps]` section (all optional). Consider the following example project file for the hypothetical application, `App`, as described earlier:

```
name = "App"
uuid = "8f986787-14fe-4607-ba5d-fbff2944afa9"

[deps]
Priv = "ba13f791-ae1d-465a-978b-69c3ad90f72b"
Pub  = "c07ecb7d-0dc9-4db7-8803-fadaaeaf08e1"
```

This project file implies the following roots map, if it was represented by a Julia dictionary:

```
roots = Dict(
    :App  => UUID("8f986787-14fe-4607-ba5d-fbff2944afa9"),
    :Priv => UUID("ba13f791-ae1d-465a-978b-69c3ad90f72b"),
    :Pub  => UUID("c07ecb7d-0dc9-4db7-8803-fadaaeaf08e1"),
)
```

Given this roots map, in App's code the statement `import Priv` will cause Julia to look up `roots[:Priv]`, which yields `ba13f791-ae1d-465a-978b-69c3ad90f72b`, the UUID of the `Priv` package that is to be loaded in that context. This UUID identifies which `Priv` package to load and use when the main application evaluates `import Priv`.

The dependency graph of a project environment is determined by the contents of the manifest file, if present. If there is no manifest file, graph is empty. A manifest file contains a stanza for each of a project's direct or indirect dependencies. For each dependency, the file lists the package's UUID and a source tree hash or an explicit path to the source code. Consider the following example manifest file for App:

```
[[Priv]] # the private one
deps = ["Pub", "Zebra"]
uuid = "ba13f791-ae1d-465a-978b-69c3ad90f72b"
path = "deps/Priv"

[[Priv]] # the public one
uuid = "2d15fe94-a1f7-436c-a4d8-07a9a496e01c"
git-tree-sha1 = "1bf63d3be994fe83456a03b874b409cfd59a6373"
version = "0.1.5"

[[Pub]]
uuid = "c07ecb7d-0dc9-4db7-8803-fadaaeaf08e1"
git-tree-sha1 = "9ebd50e2b0dd1e110e842df3b433cb5869b0dd38"
version = "2.1.4"

  [Pub.deps]
  Priv = "2d15fe94-a1f7-436c-a4d8-07a9a496e01c"
  Zebra = "f7a24cb4-21fc-4002-ac70-f0e3a0dd3f62"

[[Zebra]]
uuid = "f7a24cb4-21fc-4002-ac70-f0e3a0dd3f62"
git-tree-sha1 = "e808e36a5d7173974b90a15a353b564f3494092f"
version = "3.4.2"
```

This manifest file describes a possible complete dependency graph for the App project:

- There are two different packages named `Priv` that the application uses. It uses a private package, which is a root dependency, and a public one, which is an indirect dependency through `Pub`. These are differentiated by their distinct UUIDs, and they have different deps:
  - The private `Priv` depends on the `Pub` and `Zebra` packages.
  - The public `Priv` has no dependencies.

- The application also depends on the Pub package, which in turn depends on the public `Priv` and the same `Zebra` package that the private `Priv` package depends on.

This dependency graph represented as a dictionary, looks like this:

```
graph = Dict(
    # Priv — the private one:
    UUID("ba13f791-ae1d-465a-978b-69c3ad90f72b") => Dict(
        :Pub   => UUID("c07ecb7d-0dc9-4db7-8803-fadaaeaf08e1"),
        :Zebra => UUID("f7a24cb4-21fc-4002-ac70-f0e3a0dd3f62"),
    ),
    # Priv — the public one:
    UUID("2d15fe94-a1f7-436c-a4d8-07a9a496e01c") => Dict(),
    # Pub:
    UUID("c07ecb7d-0dc9-4db7-8803-fadaaeaf08e1") => Dict(
        :Priv  => UUID("2d15fe94-a1f7-436c-a4d8-07a9a496e01c"),
        :Zebra => UUID("f7a24cb4-21fc-4002-ac70-f0e3a0dd3f62"),
    ),
    # Zebra:
    UUID("f7a24cb4-21fc-4002-ac70-f0e3a0dd3f62") => Dict(),
)
```

Given this dependency `graph`, when Julia sees `import Priv` in the Pub package—which has UUID `c07ecb7d-0dc9-4db7-8803-fadaaeaf08e1`—it looks up:

```
graph[UUID("c07ecb7d-0dc9-4db7-8803-fadaaeaf08e1")][:Priv]
```

and gets `2d15fe94-a1f7-436c-a4d8-07a9a496e01c`, which indicates that in the context of the Pub package, `import Priv` refers to the public `Priv` package, rather than the private one which the app depends on directly. This is how the name `Priv` can refer to different packages in the main project than it does in one of its package's dependencies, which allows for duplicate names in the package ecosystem.

What happens if `import Zebra` is evaluated in the main App code base? Since `Zebra` does not appear in the project file, the import will fail even though `Zebra` *does* appear in the manifest file. Moreover, if `import Zebra` occurs in the public `Priv` package—the one with UUID `2d15fe94-a1f7-436c-a4d8-07a9a496e01c`—then that would also fail since that `Priv` package has no declared dependencies in the manifest file and therefore cannot load any packages. The `Zebra` package can only be loaded by packages for which it appears as an explicit dependency in the manifest file: the Pub package and one of the `Priv` packages.

The paths map of a project environment is extracted from the manifest file. The path of a package `uuid` named `X` is determined by these rules (in order):

1. If the project file in the directory matches `uuid` and name `X`, then either:
    - It has a toplevel `entryfile` entry, then `uuid` will be mapped to that path, interpreted relative to the directory containing the project file.
    - Otherwise, `uuid` is mapped to `src/X.jl` relative to the directory containing the project file.
2. 1. If the above is not the case and the project file has a corresponding manifest file and the manifest contains a stanza matching `uuid` then:

   * If it has a `path` entry, use that path (relative to the directory containing the manifest file).
   * If it has a `git-tree-sha1` entry, compute a deterministic hash function of `uuid` and `git-tree-sha1`—call it `slug`—and look for a directory named `packages/X/$slug` in each directory in the Julia `DEPOT_PATH` global array. Use the first such directory that exists.

2. If this is a directory then `uuid` is mapped to `src/X.jl` unless the matching manifest stanza has an `entryfile` entry in which case this is used. In both cases, these are relative to the directory in 2.1.

If any of these result in success, the path to the source code entry point will be either that result, the relative path from that result plus `src/X.jl`; otherwise, there is no path mapping for `uuid`. When loading X, if no source code path is found, the lookup will fail, and the user may be prompted to install the appropriate package version or to take other corrective action (e.g. declaring X as a dependency).

In the example manifest file above, to find the path of the first `Priv` package—the one with UUID `ba13f791-ae1d-465a-978b-69c3ad90f72b`—Julia looks for its stanza in the manifest file, sees that it has a `path` entry, looks at `deps/Priv` relative to the App project directory—let's suppose the App code lives in `/home/me/projects/App`—sees that `/home/me/projects/App/deps/Priv` exists and therefore loads `Priv` from there.

If, on the other hand, Julia was loading the *other* `Priv` package—the one with UUID `2d15fe94-a1f7-436c-a4d8-07a9a496e01c`—it finds its stanza in the manifest, see that it does *not* have a `path` entry, but that it does have a `git-tree-sha1` entry. It then computes the `slug` for this UUID/SHA-1 pair, which is `HDkrT` (the exact details of this computation aren't important, but it is consistent and deterministic). This means that the path to this `Priv` package will be `packages/Priv/HDkrT/src/Priv.jl` in one of the package depots. Suppose the contents of `DEPOT_PATH` is `["/home/me/.julia", "/usr/local/julia"]`, then Julia will look at the following paths to see if they exist:

1. `/home/me/.julia/packages/Priv/HDkrT`
2. `/usr/local/julia/packages/Priv/HDkrT`

Julia uses the first of these that exists to try to load the public `Priv` package from the file `packages/Priv/HDkrT/src/Priv.jl` in the depot where it was found.

Here is a representation of a possible paths map for our example App project environment, as provided in the Manifest given above for the dependency graph, after searching the local file system:

```
paths = Dict(
    # Priv – the private one:
    (UUID("ba13f791-ae1d-465a-978b-69c3ad90f72b"), :Priv) =>
        # relative entry-point inside `App` repo:
        "/home/me/projects/App/deps/Priv/src/Priv.jl",
    # Priv – the public one:
    (UUID("2d15fe94-a1f7-436c-a4d8-07a9a496e01c"), :Priv) =>
        # package installed in the system depot:
        "/usr/local/julia/packages/Priv/HDkrT/src/Priv.jl",
    # Pub:
    (UUID("c07ecb7d-0dc9-4db7-8803-fadaaeaf08e1"), :Pub) =>
        # package installed in the user depot:
        "/home/me/.julia/packages/Pub/oKpw/src/Pub.jl",
    # Zebra:
```

```
    (UUID("f7a24cb4-21fc-4002-ac70-f0e3a0dd3f62"), :Zebra) =>
        # package installed in the system depot:
        "/usr/local/julia/packages/Zebra/me9k/src/Zebra.jl",
)
```

This example map includes three different kinds of package locations (the first and third are part of the default load path):

1. The private `Priv` package is "vendored" inside the `App` repository.
2. The public `Priv` and `Zebra` packages are in the system depot, where packages installed and managed by the system administrator live. These are available to all users on the system.
3. The `Pub` package is in the user depot, where packages installed by the user live. These are only available to the user who installed them.

Package directories

Package directories provide a simpler kind of environment without the ability to handle name collisions. In a package directory, the set of top-level packages is the set of subdirectories that "look like" packages. A package `X` exists in a package directory if the directory contains one of the following "entry point" files:

- `X.jl`
- `X/src/X.jl`
- `X.jl/src/X.jl`

Which dependencies a package in a package directory can import depends on whether the package contains a project file:

- If it has a project file, it can only import those packages which are identified in the `[deps]` section of the project file.
- If it does not have a project file, it can import any top-level package—i.e. the same packages that can be loaded in `Main` or the REPL.

The roots map is determined by examining the contents of the package directory to generate a list of all packages that exist. Additionally, a UUID will be assigned to each entry as follows: For a given package found inside the folder `X`...

1. If `X/Project.toml` exists and has a `uuid` entry, then `uuid` is that value.
2. If `X/Project.toml` exists and but does *not* have a top-level UUID entry, `uuid` is a dummy UUID generated by hashing the canonical (real) path to `X/Project.toml`.
3. Otherwise (if `Project.toml` does not exist), then `uuid` is the all-zero nil UUID.

The dependency graph of a project directory is determined by the presence and contents of project files in the subdirectory of each package. The rules are:

- If a package subdirectory has no project file, then it is omitted from graph and import statements in its code are treated as top-level, the same as the main project and REPL.
- If a package subdirectory has a project file, then the graph entry for its UUID is the `[deps]` map of the project file, which is considered to be empty if the section is absent.

As an example, suppose a package directory has the following structure and content:

```
Aardvark/
    src/Aardvark.jl:
        import Bobcat
        import Cobra

Bobcat/
    Project.toml:
        [deps]
        Cobra = "4725e24d-f727-424b-bca0-c4307a3456fa"
        Dingo = "7a7925be-828c-4418-bbeb-bac8dfc843bc"

    src/Bobcat.jl:
        import Cobra
        import Dingo

Cobra/
    Project.toml:
        uuid = "4725e24d-f727-424b-bca0-c4307a3456fa"
        [deps]
        Dingo = "7a7925be-828c-4418-bbeb-bac8dfc843bc"

    src/Cobra.jl:
        import Dingo

Dingo/
    Project.toml:
        uuid = "7a7925be-828c-4418-bbeb-bac8dfc843bc"

    src/Dingo.jl:
        # no imports
```

Here is a corresponding roots structure, represented as a dictionary:

```
roots = Dict(
    :Aardvark => UUID("00000000-0000-0000-0000-000000000000"), # no project file, nil UUID
    :Bobcat   => UUID("85ad11c7-31f6-5d08-84db-0a4914d4cadf"), # dummy UUID based on path
    :Cobra    => UUID("4725e24d-f727-424b-bca0-c4307a3456fa"), # UUID from project file
    :Dingo    => UUID("7a7925be-828c-4418-bbeb-bac8dfc843bc"), # UUID from project file
)
```

Here is the corresponding graph structure, represented as a dictionary:

```
graph = Dict(
    # Bobcat:
    UUID("85ad11c7-31f6-5d08-84db-0a4914d4cadf") => Dict(
        :Cobra => UUID("4725e24d-f727-424b-bca0-c4307a3456fa"),
        :Dingo => UUID("7a7925be-828c-4418-bbeb-bac8dfc843bc"),
    ),
    # Cobra:
    UUID("4725e24d-f727-424b-bca0-c4307a3456fa") => Dict(
        :Dingo => UUID("7a7925be-828c-4418-bbeb-bac8dfc843bc"),
    ),
    # Dingo:
    UUID("7a7925be-828c-4418-bbeb-bac8dfc843bc") => Dict(),
)
```

A few general rules to note:

1. A package without a project file can depend on any top-level dependency, and since every package in a package directory is available at the top-level, it can import all packages in the environment.
2. A package with a project file cannot depend on one without a project file since packages with project files can only load packages in `graph` and packages without project files do not appear in `graph`.
3. A package with a project file but no explicit UUID can only be depended on by packages without project files since dummy UUIDs assigned to these packages are strictly internal.

Observe the following specific instances of these rules in our example:

- `Aardvark` can import on any of `Bobcat`, `Cobra` or `Dingo`; it does import `Bobcat` and `Cobra`.
- `Bobcat` can and does import both `Cobra` and `Dingo`, which both have project files with UUIDs and are declared as dependencies in `Bobcat`'s `[deps]` section.
- `Bobcat` cannot depend on `Aardvark` since `Aardvark` does not have a project file.
- `Cobra` can and does import `Dingo`, which has a project file and UUID, and is declared as a dependency in `Cobra`'s `[deps]` section.
- `Cobra` cannot depend on `Aardvark` or `Bobcat` since neither have real UUIDs.
- `Dingo` cannot import anything because it has a project file without a `[deps]` section.

The paths map in a package directory is simple: it maps subdirectory names to their corresponding entry-point paths. In other words, if the path to our example project directory is `/home/me/animals` then the `paths` map could be represented by this dictionary:

```
paths = Dict(
    (UUID("00000000-0000-0000-0000-000000000000"), :Aardvark) =>
        "/home/me/AnimalPackages/Aardvark/src/Aardvark.jl",
    (UUID("85ad11c7-31f6-5d08-84db-0a4914d4cadf"), :Bobcat) =>
        "/home/me/AnimalPackages/Bobcat/src/Bobcat.jl",
    (UUID("4725e24d-f727-424b-bca0-c4307a3456fa"), :Cobra) =>
```

```
        "/home/me/AnimalPackages/Cobra/src/Cobra.jl",
    (UUID("7a7925be-828c-4418-bbeb-bac8dfc843bc"), :Dingo) =>
        "/home/me/AnimalPackages/Dingo/src/Dingo.jl",
)
```

Since all packages in a package directory environment are, by definition, subdirectories with the expected entry-point files, their `paths` map entries always have this form.

Environment stacks

The third and final kind of environment is one that combines other environments by overlaying several of them, making the packages in each available in a single composite environment. These composite environments are called *environment stacks*. The Julia `LOAD_PATH` global defines an environment stack—the environment in which the Julia process operates. If you want your Julia process to have access only to the packages in one project or package directory, make it the only entry in `LOAD_PATH`. It is often quite useful, however, to have access to some of your favorite tools—standard libraries, profilers, debuggers, personal utilities, etc.—even if they are not dependencies of the project you're working on. By adding an environment containing these tools to the load path, you immediately have access to them in top-level code without needing to add them to your project.

The mechanism for combining the roots, graph and paths data structures of the components of an environment stack is simple: they are merged as dictionaries, favoring earlier entries over later ones in the case of key collisions. In other words, if we have `stack = [env₁, env₂, …]` then we have:

```
roots = reduce(merge, reverse([roots₁, roots₂, …]))
graph = reduce(merge, reverse([graph₁, graph₂, …]))
paths = reduce(merge, reverse([paths₁, paths₂, …]))
```

The subscripted `rootsᵢ`, `graphᵢ` and `pathsᵢ` variables correspond to the subscripted environments, `envᵢ`, contained in `stack`. The `reverse` is present because `merge` favors the last argument rather than first when there are collisions between keys in its argument dictionaries. There are a couple of noteworthy features of this design:

1. The *primary environment*—i.e. the first environment in a stack—is faithfully embedded in a stacked environment. The full dependency graph of the first environment in a stack is guaranteed to be included intact in the stacked environment including the same versions of all dependencies.

2. Packages in non-primary environments can end up using incompatible versions of their dependencies even if their own environments are entirely compatible. This can happen when one of their dependencies is shadowed by a version in an earlier environment in the stack (either by graph or path, or both).

Since the primary environment is typically the environment of a project you're working on, while environments later in the stack contain additional tools, this is the right trade-off: it's better to break your development tools but keep the project working. When such incompatibilities occur, you'll typically want to upgrade your dev tools to versions that are compatible with the main project.

Package Extensions

A package "extension" is a module that is automatically loaded when a specified set of other packages (its "triggers") are loaded in the current Julia session. Extensions are defined under the `[extensions]` section in the project file. The triggers of an extension are a subset of those packages listed under the `[weakdeps]` (and possibly, but uncommonly the `[deps]`) section of the project file. Those packages can have compat entries like other packages.

```
name = "MyPackage"

[compat]
ExtDep = "1.0"
OtherExtDep = "1.0"

[weakdeps]
ExtDep = "c9a23..." # uuid
OtherExtDep = "862e..." # uuid

[extensions]
BarExt = ["ExtDep", "OtherExtDep"]
FooExt = "ExtDep"
...
```

The keys under `extensions` are the names of the extensions. They are loaded when all the packages on the right hand side (the triggers) of that extension are loaded. If an extension only has one trigger the list of triggers can be written as just a string for brevity. The location for the entry point of the extension is either in `ext/FooExt.jl` or `ext/FooExt/FooExt.jl` for extension `FooExt`. The content of an extension is often structured as:

```
module FooExt

# Load main package and triggers
using MyPackage, ExtDep

# Extend functionality in main package with types from the triggers
MyPackage.func(x::ExtDep.SomeStruct) = ...

end
```

When a package with extensions is added to an environment, the `weakdeps` and `extensions` sections are stored in the manifest file in the section for that package. The dependency lookup rules for a package are the same as for its "parent" except that the listed triggers are also considered as dependencies.

Workspaces

A project file can define a workspace by giving a set of projects that is part of that workspace:

```
[workspace]
projects = ["test", "benchmarks", "docs", "SomePackage"]
```

Each project listed in the `projects` array is specified by its relative path from the workspace root. This can be a direct child directory (e.g., `"test"`) or a nested subdirectory (e.g., `"nested/subdir/MyPackage"`). Each project contains its own `Project.toml` file, which may include additional dependencies and compatibility constraints. In such cases, the package manager gathers all dependency information from all the projects in the workspace generating a single manifest file that combines the versions of all dependencies.

When Julia loads a project, it searches upward through parent directories until it reaches the user's home directory to find a workspace that includes that project. This allows workspace projects to be nested at arbitrary depth within the workspace directory tree.

Furthermore, workspaces can be "nested", meaning a project defining a workspace can also be part of another workspace. In this scenario, a single manifest file is still utilized, stored alongside the "root project" (the project that doesn't have another workspace including it). An example file structure could look like this:

```
Project.toml # projects = ["MyPackage"]
Manifest.toml
MyPackage/
    Project.toml # projects = ["test"]
    test/
        Project.toml
```

Package/Environment Preferences

Preferences are dictionaries of metadata that influence package behavior within an environment. The preferences system supports reading preferences at compile-time, which means that at code-loading time, we must ensure that the precompilation files selected by Julia were built with the same preferences as the current environment before loading them. The public API for modifying Preferences is contained within the Preferences.jl package. Preferences are stored as TOML dictionaries within a `(Julia)LocalPreferences.toml` file next to the currently-active project. If a preference is "exported", it is instead stored within the `(Julia)Project.toml` instead. The intention is to allow shared projects to contain shared preferences, while allowing for users themselves to override those preferences with their own settings in the LocalPreferences.toml file, which should be .gitignored as the name implies.

Preferences that are accessed during compilation are automatically marked as compile-time preferences, and any change recorded to these preferences will cause the Julia compiler to recompile any cached precompilation file(s) (`.ji` and corresponding `.so`, `.dll`, or `.dylib` files) for that module. This is done by serializing the hash of all compile-time preferences during compilation, then checking that hash against the current environment when searching for the proper file(s) to load.

Preferences can be set with depot-wide defaults; if package Foo is installed within your global environment and it has preferences set, these preferences will apply as long as your global environment is part of your `LOAD_PATH`. Preferences in environments higher up in the environment stack get overridden by the more proximal entries in the load path, ending with the currently active project. This allows depot-wide preference defaults to exist, with active projects able to merge or even completely overwrite these inherited preferences. See the docstring for `Preferences.set_preferences!()` for the full details of how to set preferences to allow or disallow merging.

32.4 Conclusion

Federated package management and precise software reproducibility are difficult but worthy goals in a package system. In combination, these goals lead to a more complex package loading mechanism than

most dynamic languages have, but it also yields scalability and reproducibility that is more commonly associated with static languages. Typically, Julia users should be able to use the built-in package manager to manage their projects without needing a precise understanding of these interactions. A call to `Pkg.add("X")` will add to the appropriate project and manifest files, selected via `Pkg.activate("Y")`, so that a future call to `import X` will load X without further thought.

Chapter 33

Profiling

The `Profile` module provides tools to help developers improve the performance of their code. When used, it takes measurements on running code, and produces output that helps you understand how much time is spent on individual line(s). The most common usage is to identify "bottlenecks" as targets for optimization.

`Profile` implements what is known as a "sampling" or statistical profiler. It works by periodically taking a backtrace during the execution of any task. Each backtrace captures the currently-running function and line number, plus the complete chain of function calls that led to this line, and hence is a "snapshot" of the current state of execution.

If much of your run time is spent executing a particular line of code, this line will show up frequently in the set of all backtraces. In other words, the "cost" of a given line–or really, the cost of the sequence of function calls up to and including this line–is proportional to how often it appears in the set of all backtraces.

A sampling profiler does not provide complete line-by-line coverage, because the backtraces occur at intervals (by default, 1 ms on Unix systems and 10 ms on Windows, although the actual scheduling is subject to operating system load). Moreover, as discussed further below, because samples are collected at a sparse subset of all execution points, the data collected by a sampling profiler is subject to statistical noise.

Despite these limitations, sampling profilers have substantial strengths:

- You do not have to make any modifications to your code to take timing measurements.
- It can profile into Julia's core code and even (optionally) into C and Fortran libraries.
- By running "infrequently" there is very little performance overhead; while profiling, your code can run at nearly native speed.

For these reasons, it's recommended that you try using the built-in sampling profiler before considering any alternatives.

33.1 Basic usage

Let's work with a simple test case:

```
julia> function myfunc()
```

```
        A = rand(200, 200, 400)
        maximum(A)
    end
```

It's a good idea to first run the code you intend to profile at least once (unless you want to profile Julia's JIT-compiler):

```
julia> myfunc() # run once to force compilation
```

Now we're ready to profile this function:

```
julia> using Profile

julia> @profile myfunc()
```

To see the profiling results, there are several graphical browsers. One "family" of visualizers is based on FlameGraphs.jl, with each family member providing a different user interface:

- VS Code is a full IDE with built-in support for profile visualization
- ProfileView.jl is a stand-alone visualizer based on GTK
- ProfileVega.jl uses VegaLight and integrates well with Jupyter notebooks
- StatProfilerHTML.jl produces HTML and presents some additional summaries, and also integrates well with Jupyter notebooks
- ProfileSVG.jl renders SVG
- PProf.jl serves a local website for inspecting graphs, flamegraphs and more
- ProfileCanvas.jl is a HTML canvas based profile viewer UI, used by the Julia VS Code extension, but can also generate interactive HTML files.

An entirely independent approach to profile visualization is PProf.jl, which uses the external `pprof` tool.

Here, though, we'll use the text-based display that comes with the standard library:

```
julia> Profile.print()
80 ./event.jl:73; (::Base.REPL.##1#2{Base.REPL.REPLBackend})()
 80 ./REPL.jl:97; macro expansion
  80 ./REPL.jl:66; eval_user_input(::Any, ::Base.REPL.REPLBackend)
   80 ./boot.jl:235; eval(::Module, ::Any)
    80 ./<missing>:?; anonymous
     80 ./profile.jl:23; macro expansion
      52 ./REPL[1]:2; myfunc()
       38 ./random.jl:431; rand!(::MersenneTwister, ::Array{Float64,3}, ::Int64, ::Type{B...
        38 ./dSFMT.jl:84; dsfmt_fill_array_close_open!(::Base.dSFMT.DSFMT_state, ::Ptr{F...
       14 ./random.jl:278; rand
        14 ./random.jl:277; rand
         14 ./random.jl:366; rand
```

```
         14 ./random.jl:369; rand
      28 ./REPL[1]:3; myfunc()
       28 ./reduce.jl:270; _mapreduce(::Base.#identity, ::Base.#scalarmax, ::IndexLinear,...
        3  ./reduce.jl:426; mapreduce_impl(::Base.#identity, ::Base.#scalarmax, ::Array{F...
        25 ./reduce.jl:428; mapreduce_impl(::Base.#identity, ::Base.#scalarmax, ::Array{F...
```

Each line of this display represents a particular spot (line number) in the code. Indentation is used to indicate the nested sequence of function calls, with more-indented lines being deeper in the sequence of calls. In each line, the first "field" is the number of backtraces (samples) taken *at this line or in any functions executed by this line*. The second field is the file name and line number and the third field is the function name. Note that the specific line numbers may change as Julia's code changes; if you want to follow along, it's best to run this example yourself.

In this example, we can see that the top level function called is in the file `event.jl`. This is the function that runs the REPL when you launch Julia. If you examine line 97 of `REPL.jl`, you'll see this is where the function `eval_user_input()` is called. This is the function that evaluates what you type at the REPL, and since we're working interactively these functions were invoked when we entered `@profile myfunc()`. The next line reflects actions taken in the `@profile` macro.

The first line shows that 80 backtraces were taken at line 73 of `event.jl`, but it's not that this line was "expensive" on its own: the third line reveals that all 80 of these backtraces were actually triggered inside its call to `eval_user_input`, and so on. To find out which operations are actually taking the time, we need to look deeper in the call chain.

The first "important" line in this output is this one:

```
52 ./REPL[1]:2; myfunc()
```

REPL refers to the fact that we defined `myfunc` in the REPL, rather than putting it in a file; if we had used a file, this would show the file name. The `[1]` shows that the function `myfunc` was the first expression evaluated in this REPL session. Line 2 of `myfunc()` contains the call to `rand`, and there were 52 (out of 80) backtraces that occurred at this line. Below that, you can see a call to `dsfmt_fill_array_close_open!` inside `dSFMT.jl`.

A little further down, you see:

```
28 ./REPL[1]:3; myfunc()
```

Line 3 of `myfunc` contains the call to `maximum`, and there were 28 (out of 80) backtraces taken here. Below that, you can see the specific places in `base/reduce.jl` that carry out the time-consuming operations in the `maximum` function for this type of input data.

Overall, we can tentatively conclude that generating the random numbers is approximately twice as expensive as finding the maximum element. We could increase our confidence in this result by collecting more samples:

```
julia> @profile (for i = 1:100; myfunc(); end)

julia> Profile.print()
[....]
 3821 ./REPL[1]:2; myfunc()
```

```
 3511 ./random.jl:431; rand!(::MersenneTwister, ::Array{Float64,3}, ::Int64, ::Type...
  3511 ./dSFMT.jl:84; dsfmt_fill_array_close_open!(::Base.dSFMT.DSFMT_state, ::Ptr...
 310  ./random.jl:278; rand
  [....]
2893 ./REPL[1]:3; myfunc()
 2893 ./reduce.jl:270; _mapreduce(::Base.#identity, ::Base.#scalarmax, ::IndexLinea...
  [....]
```

In general, if you have `N` samples collected at a line, you can expect an uncertainty on the order of `sqrt(N)` (barring other sources of noise, like how busy the computer is with other tasks). The major exception to this rule is garbage collection, which runs infrequently but tends to be quite expensive. (Since Julia's garbage collector is written in C, such events can be detected using the `C=true` output mode described below, or by using ProfileView.jl.)

This illustrates the default "tree" dump; an alternative is the "flat" dump, which accumulates counts independent of their nesting:

```
julia> Profile.print(format=:flat)
 Count File          Line Function
  6714 ./<missing>     -1 anonymous
  6714 ./REPL.jl       66 eval_user_input(::Any, ::Base.REPL.REPLBackend)
  6714 ./REPL.jl       97 macro expansion
  3821 ./REPL[1]        2 myfunc()
  2893 ./REPL[1]        3 myfunc()
  6714 ./REPL[7]        1 macro expansion
  6714 ./boot.jl      235 eval(::Module, ::Any)
  3511 ./dSFMT.jl      84 dsfmt_fill_array_close_open!(::Base.dSFMT.DSFMT_s...
  6714 ./event.jl      73 (::Base.REPL.##1#2{Base.REPL.REPLBackend})()
  6714 ./profile.jl    23 macro expansion
  3511 ./random.jl    431 rand!(::MersenneTwister, ::Array{Float64,3}, ::In...
   310 ./random.jl    277 rand
   310 ./random.jl    278 rand
   310 ./random.jl    366 rand
   310 ./random.jl    369 rand
  2893 ./reduce.jl    270 _mapreduce(::Base.#identity, ::Base.#scalarmax, :...
     5 ./reduce.jl    420 mapreduce_impl(::Base.#identity, ::Base.#scalarma...
   253 ./reduce.jl    426 mapreduce_impl(::Base.#identity, ::Base.#scalarma...
  2592 ./reduce.jl    428 mapreduce_impl(::Base.#identity, ::Base.#scalarma...
    43 ./reduce.jl    429 mapreduce_impl(::Base.#identity, ::Base.#scalarma...
```

If your code has recursion, one potentially-confusing point is that a line in a "child" function can accumulate more counts than there are total backtraces. Consider the following function definitions:

```
dumbsum(n::Integer) = n == 1 ? 1 : 1 + dumbsum(n-1)
dumbsum3() = dumbsum(3)
```

If you were to profile `dumbsum3`, and a backtrace was taken while it was executing `dumbsum(1)`, the backtrace would look like this:

```
dumbsum3
```

```
    dumbsum(3)
        dumbsum(2)
            dumbsum(1)
```

Consequently, this child function gets 3 counts, even though the parent only gets one. The "tree" representation makes this much clearer, and for this reason (among others) is probably the most useful way to view the results.

33.2 Accumulation and clearing

Results from `@profile` accumulate in a buffer; if you run multiple pieces of code under `@profile`, then `Profile.print()` will show you the combined results. This can be very useful, but sometimes you want to start fresh; you can do so with `Profile.clear()`.

33.3 Options for controlling the display of profile results

`Profile.print` has more options than we've described so far. Let's see the full declaration:

```
function print(io::IO = stdout, data = fetch(); kwargs...)
```

Let's first discuss the two positional arguments, and later the keyword arguments:

- `io` – Allows you to save the results to a buffer, e.g. a file, but the default is to print to `stdout` (the console).
- `data` – Contains the data you want to analyze; by default that is obtained from `Profile.fetch()`, which pulls out the backtraces from a pre-allocated buffer. For example, if you want to profile the profiler, you could say:

  ```
  data = copy(Profile.fetch())
  Profile.clear()
  @profile Profile.print(stdout, data) # Prints the previous results
  Profile.print()                      # Prints results from Profile.print()
  ```

The keyword arguments can be any combination of:

- `format` – Introduced above, determines whether backtraces are printed with (default, `:tree`) or without (`:flat`) indentation indicating tree structure.
- `C` – If `true`, backtraces from C and Fortran code are shown (normally they are excluded). Try running the introductory example with `Profile.print(C = true)`. This can be extremely helpful in deciding whether it's Julia code or C code that is causing a bottleneck; setting `C = true` also improves the interpretability of the nesting, at the cost of longer profile dumps.
- `combine` – Some lines of code contain multiple operations; for example, `s += A[i]` contains both an array reference (`A[i]`) and a sum operation. These correspond to different lines in the generated machine code, and hence there may be two or more different addresses captured during backtraces on this line. `combine = true` lumps them together, and is probably what you typically want, but you can generate an output separately for each unique instruction pointer with `combine = false`.

- `maxdepth` – Limits frames at a depth higher than `maxdepth` in the `:tree` format.
- `sortedby` – Controls the order in `:flat` format. `:filefuncline` (default) sorts by the source line, whereas `:count` sorts in order of number of collected samples.
- `noisefloor` – Limits frames that are below the heuristic noise floor of the sample (only applies to format `:tree`). A suggested value to try for this is 2.0 (the default is 0). This parameter hides samples for which `n <= noisefloor * √N`, where `n` is the number of samples on this line, and `N` is the number of samples for the callee.
- `mincount` – Limits frames with less than `mincount` occurrences.

File/function names are sometimes truncated (with `...`), and indentation is truncated with a `+n` at the beginning, where `n` is the number of extra spaces that would have been inserted, had there been room. If you want a complete profile of deeply-nested code, often a good idea is to save to a file using a wide `displaysize` in an `IOContext`:

```
open("/tmp/prof.txt", "w") do s
    Profile.print(IOContext(s, :displaysize => (24, 500)))
end
```

33.4 Configuration

`@profile` just accumulates backtraces, and the analysis happens when you call `Profile.print()`. For a long-running computation, it's entirely possible that the pre-allocated buffer for storing backtraces will be filled. If that happens, the backtraces stop but your computation continues. As a consequence, you may miss some important profiling data (you will get a warning when that happens).

You can obtain and configure the relevant parameters this way:

```
Profile.init() # returns the current settings
Profile.init(n = 10^7, delay = 0.01)
```

`n` is the total number of instruction pointers you can store, with a default value of `10^6`. If your typical backtrace is 20 instruction pointers, then you can collect 50000 backtraces, which suggests a statistical uncertainty of less than 1%. This may be good enough for most applications.

Consequently, you are more likely to need to modify `delay`, expressed in seconds, which sets the amount of time that Julia gets between snapshots to perform the requested computations. A very long-running job might not need frequent backtraces. The default setting is `delay = 0.001`. Of course, you can decrease the delay as well as increase it; however, the overhead of profiling grows once the delay becomes similar to the amount of time needed to take a backtrace (~30 microseconds on the author's laptop).

33.5 Wall-time Profiler

Introduction & Problem Motivation

The profiler described in the previous section is a sampling CPU profiler. At a high level, the profiler periodically stops all Julia compute threads to collect their backtraces and estimates the time spent in each function based on the number of backtrace samples that include a frame from that function. However,

note that only tasks currently running on system threads just before the profiler stops them will have their backtraces collected.

While this profiler is typically well-suited for workloads where the majority of tasks are compute-bound, it is less helpful for systems where most tasks are IO-heavy or for diagnosing contention on synchronization primitives in your code.

Let's consider this simple workload:

```
using Base.Threads
using Profile
using PProf

ch = Channel(1)

const N_SPAWNED_TASKS = (1 << 10)
const WAIT_TIME_NS = 10_000_000

function spawn_a_bunch_of_tasks_waiting_on_channel()
    for i in 1:N_SPAWNED_TASKS
        Threads.@spawn begin
            take!(ch)
        end
    end
end

function busywait()
    t0 = time_ns()
    while true
        if time_ns() - t0 > WAIT_TIME_NS
            break
        end
    end
end

function main()
    spawn_a_bunch_of_tasks_waiting_on_channel()
    for i in 1:N_SPAWNED_TASKS
        put!(ch, i)
        busywait()
    end
end

Profile.@profile main()
```

Our goal is to detect whether there is contention on the `ch` channel—i.e., whether the number of waiters is excessive given the rate at which work items are being produced in the channel.

If we run this, we obtain the following PProf flame graph:

This profile provides no information to help determine where contention occurs in the system's synchronization primitives. Waiters on a channel will be blocked and descheduled, meaning no system thread will be running the tasks assigned to those waiters, and as a result, they won't be sampled by the profiler.

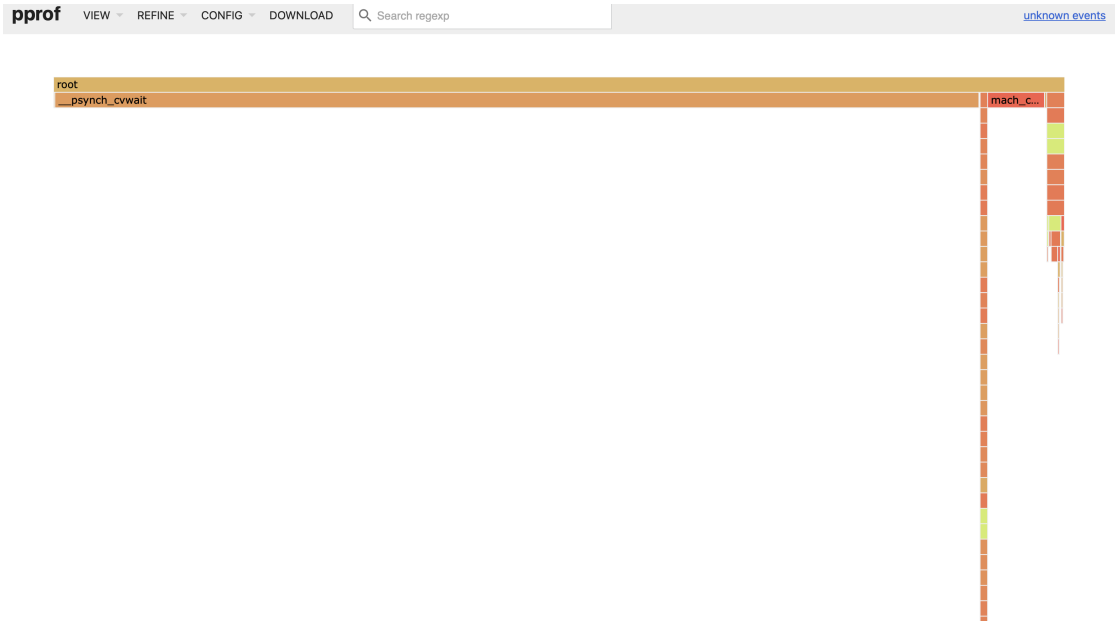


Figure 33.1: CPU Profile

Wall-time Profiler

Instead of sampling threads—and thus only sampling tasks that are running—a wall-time task profiler samples tasks independently of their scheduling state. For example, tasks that are sleeping on a synchronization primitive at the time the profiler is running will be sampled with the same probability as tasks that were actively running when the profiler attempted to capture backtraces.

This approach allows us to construct a profile where backtraces from tasks blocked on the `ch` channel, as in the example above, are actually represented.

Let's run the same example, but now with a wall-time profiler:

```
using Base.Threads
using Profile
using PProf

ch = Channel(1)

const N_SPAWNED_TASKS = (1 << 10)
const WAIT_TIME_NS = 10_000_000

function spawn_a_bunch_of_tasks_waiting_on_channel()
    for i in 1:N_SPAWNED_TASKS
        Threads.@spawn begin
            take!(ch)
        end
    end
end
```

Figure 33.2: Wall-time Profile Channel

```
function busywait()
    t0 = time_ns()
    while true
        if time_ns() - t0 > WAIT_TIME_NS
            break
        end
    end
end

function main()
    spawn_a_bunch_of_tasks_waiting_on_channel()
    for i in 1:N_SPAWNED_TASKS
        put!(ch, i)
        busywait()
    end
end

Profile.@profile_walltime main()
```

We obtain the following flame graph:

We see that a large number of samples come from channel-related `take!` functions, which allows us to determine that there is indeed an excessive number of waiters in `ch`.

A Compute-Bound Workload

Despite the wall-time profiler sampling all live tasks in the system and not just the currently running ones, it can still be helpful for identifying performance hotspots, even if your code is compute-bound. Let's consider a simple example:

```
using Base.Threads
using Profile
using PProf

ch = Channel(1)
```

```
const MAX_ITERS = (1 << 22)
const N_TASKS = (1 << 12)

function spawn_a_task_waiting_on_channel()
    Threads.@spawn begin
        take!(ch)
    end
end

function sum_of_sqrt()
    sum_of_sqrt = 0.0
    for i in 1:MAX_ITERS
        sum_of_sqrt += sqrt(i)
    end
    return sum_of_sqrt
end

function spawn_a_bunch_of_compute_heavy_tasks()
    Threads.@sync begin
        for i in 1:N_TASKS
            Threads.@spawn begin
                sum_of_sqrt()
            end
        end
    end
end

function main()
    spawn_a_task_waiting_on_channel()
    spawn_a_bunch_of_compute_heavy_tasks()
end

Profile.@profile_walltime main()
```

After collecting a wall-time profile, we get the following flame graph:

Notice how many of the samples contain `sum_of_sqrt`, which is the expensive compute function in our example.

Identifying Task Sampling Failures in your Profile

In the current implementation, the wall-time profiler attempts to sample from tasks that have been alive since the last garbage collection, along with those created afterward. However, if most tasks are extremely short-lived, you may end up sampling tasks that have already completed, resulting in missed backtrace captures.

If you encounter samples containing `failed_to_sample_task_fun` or `failed_to_stop_thread_fun`, this likely indicates a high volume of short-lived tasks, which prevented their backtraces from being collected.

Let's consider this simple example:

```
using Base.Threads
```

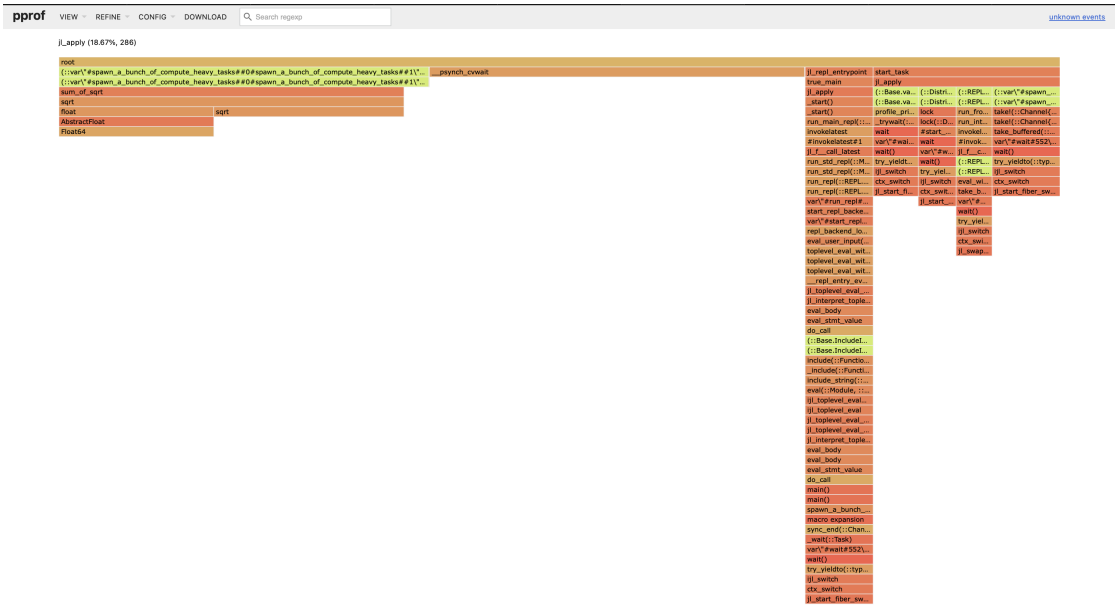


Figure 33.3: Wall-time Profile Compute-Bound

```
using Profile
using PProf

const N_SPAWNED_TASKS = (1 << 16)
const WAIT_TIME_NS = 100_000

function spawn_a_bunch_of_short_lived_tasks()
    for i in 1:N_SPAWNED_TASKS
        Threads.@spawn begin
            # Do nothing
        end
    end
end

function busywait()
    t0 = time_ns()
    while true
        if time_ns() - t0 > WAIT_TIME_NS
            break
        end
    end
end

function main()
    GC.enable(false)
    spawn_a_bunch_of_short_lived_tasks()
    for i in 1:N_SPAWNED_TASKS
        busywait()
```

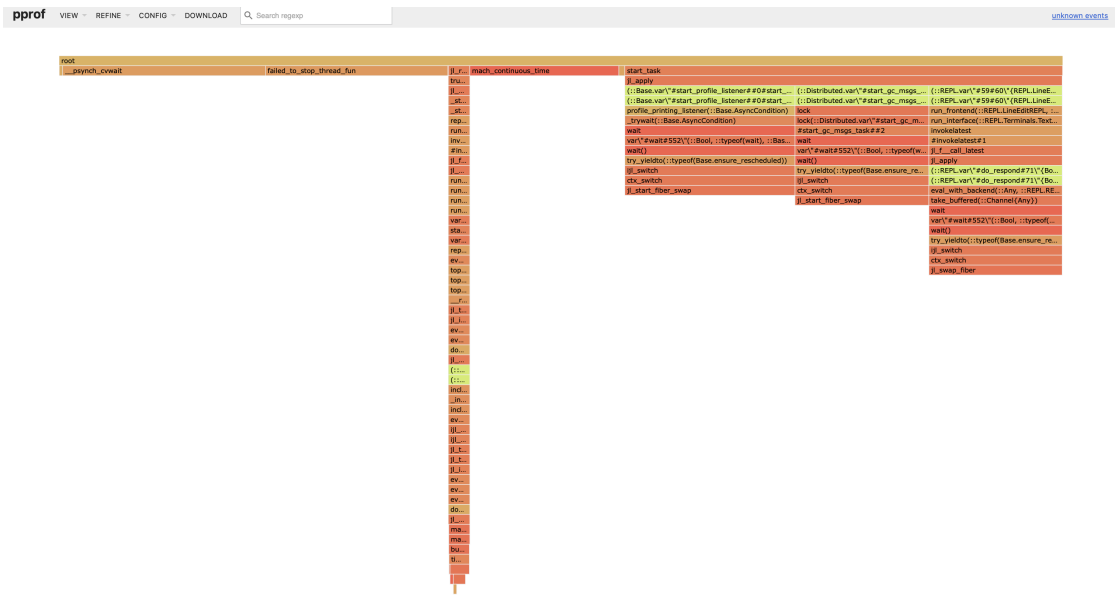


Figure 33.4: Task Sampling Failure

```
    end
    GC.enable(true)
end

Profile.@profile_walltime main()
```

Notice that the tasks spawned in `spawn_a_bunch_of_short_lived_tasks` are extremely short-lived. Since these tasks constitute the majority in the system, we will likely miss capturing a backtrace for most sampled tasks.

After collecting a wall-time profile, we obtain the following flame graph:

The large number of samples from `failed_to_stop_thread_fun` confirms that we have a significant number of short-lived tasks in the system.

33.6 Memory allocation analysis

One of the most common techniques to improve performance is to reduce memory allocation. Julia provides several tools to measure this:

`@time`

The total amount of allocation can be measured with `@time`, `@allocated` and `@allocations`, and specific lines triggering allocation can often be inferred from profiling via the cost of garbage collection that these lines incur. However, sometimes it is more efficient to directly measure the amount of memory allocated by each line of code.

GC Logging

While `@time` logs high-level stats about memory usage and garbage collection over the course of evaluating an expression, it can be useful to log each garbage collection event, to get an intuitive sense of how often the garbage collector is running, how long it's running each time, and how much garbage it collects each time. This can be enabled with `GC.enable_logging(true)`, which causes Julia to log to stderr every time a garbage collection happens.

Allocation Profiler

Julia 1.8

This functionality requires at least Julia 1.8.

The allocation profiler records the stack trace, type, and size of each allocation while it is running. It can be invoked with `Profile.Allocs.@profile`.

This information about the allocations is returned as an array of `Alloc` objects, wrapped in an `AllocResults` object. The best way to visualize these is currently with the PProf.jl and ProfileCanvas.jl packages, which can visualize the call stacks which are making the most allocations.

The allocation profiler does have significant overhead, so a `sample_rate` argument can be passed to speed it up by making it skip some allocations. Passing `sample_rate=1.0` will make it record everything (which is slow); `sample_rate=0.1` will record only 10% of the allocations (faster), etc.

Julia 1.11

Older versions of Julia could not capture types in all cases. In older versions of Julia, if you see an allocation of type `Profile.Allocs.UnknownType`, it means that the profiler doesn't know what type of object was allocated. This mainly happened when the allocation was coming from generated code produced by the compiler. See issue #43688 for more info.

Since Julia 1.11, all allocations should have a type reported.

For more details on how to use this tool, please see the talk from JuliaCon 2022.

Allocation Profiler Example

In this simple example, we use PProf to visualize the alloc profile. You could use another visualization tool instead. We collect the profile (specifying a sample rate), then we visualize it.

```
using Profile, PProf
Profile.Allocs.clear()
Profile.Allocs.@profile sample_rate=0.0001 my_function()
PProf.Allocs.pprof()
```

Here is a more in-depth example, showing how we can tune the sample rate. A good number of samples to aim for is around 1 - 10 thousand. Too many, and the profile visualizer can get overwhelmed, and profiling will be slow. Too few, and you don't have a representative sample.

```
julia> import Profile
```

```
julia> @time my_function()  # Estimate allocations from a (second-run) of the function
  0.110018 seconds (1.50 M allocations: 58.725 MiB, 17.17% gc time)
500000

julia> Profile.Allocs.clear()

julia> Profile.Allocs.@profile sample_rate=0.001 begin   # 1.5 M * 0.001 = ~1.5K allocs.
           my_function()
       end
500000

julia> prof = Profile.Allocs.fetch();  # If you want, you can also manually inspect the results.

julia> length(prof.allocs)  # Confirm we have expected number of allocations.
1515

julia> using PProf  # Now, visualize with an external tool, like PProf or ProfileCanvas.

julia> PProf.Allocs.pprof(prof; from_c=false)  # You can optionally pass in a previously fetched
↪  profile result.
Analyzing 1515 allocation samples... 100%|████████████████████████████████| Time: 0:00:00
Main binary filename not available.
Serving web UI on http://localhost:62261
"alloc-profile.pb.gz"
```

Then you can view the profile by navigating to http://localhost:62261, and the profile is saved to disk. See PProf package for more options.

Allocation Profiling Tips

As stated above, aim for around 1-10 thousand samples in your profile.

Note that we are uniformly sampling in the space of *all allocations*, and are not weighting our samples by the size of the allocation. So a given allocation profile may not give a representative profile of where most bytes are allocated in your program, unless you had set `sample_rate=1`.

Allocations can come from users directly constructing objects, but can also come from inside the runtime or be inserted into compiled code to handle type instability. Looking at the "source code" view can be helpful to isolate them, and then other external tools such as `Cthulhu.jl` can be useful for identifying the cause of the allocation.

Allocation Profile Visualization Tools

There are several profiling visualization tools now that can all display Allocation Profiles. Here is a small list of some of the main ones we know about:

- PProf.jl
- ProfileCanvas.jl
- VSCode's built-in profile visualizer (`@profview_allocs`) [docs needed]
- Viewing the results directly in the REPL

- You can inspect the results in the REPL via `Profile.Allocs.fetch()`, to view the stacktrace and type of each allocation.

Line-by-Line Allocation Tracking

An alternative way to measure allocations is to start Julia with the `--track-allocation=<setting>` command-line option, for which you can choose `none` (the default, do not measure allocation), `user` (measure memory allocation everywhere except Julia's core code), or `all` (measure memory allocation at each line of Julia code). Allocation gets measured for each line of compiled code. When you quit Julia, the cumulative results are written to text files with `.mem` appended after the file name, residing in the same directory as the source file. Each line lists the total number of bytes allocated. The `Coverage` package contains some elementary analysis tools, for example to sort the lines in order of number of bytes allocated.

In interpreting the results, there are a few important details. Under the `user` setting, the first line of any function directly called from the REPL will exhibit allocation due to events that happen in the REPL code itself. More significantly, JIT-compilation also adds to allocation counts, because much of Julia's compiler is written in Julia (and compilation usually requires memory allocation). The recommended procedure is to force compilation by executing all the commands you want to analyze, then call `Profile.clear_malloc_data()` to reset all allocation counters. Finally, execute the desired commands and quit Julia to trigger the generation of the `.mem` files.

Note

`--track-allocation` changes code generation to log the allocations, and so the allocations may be different than what happens without the option. We recommend using the allocation profiler instead.

33.7 External Profiling

Currently Julia supports `Intel VTune`, `OProfile` and `perf` as external profiling tools.

Depending on the tool you choose, compile with `USE_INTEL_JITEVENTS`, `USE_OPROFILE_JITEVENTS` and `USE_PERF_JITEVENTS` set to 1 in `Make.user`. Multiple flags are supported.

Before running Julia set the environment variable `ENABLE_JITPROFILING` to 1.

Now you have a multitude of ways to employ those tools! For example with `OProfile` you can try a simple recording :

```
>ENABLE_JITPROFILING=1 sudo operf -Vdebug ./julia test/fastmath.jl
>opreport -l `which ./julia`
```

Or similarly with `perf` :

```
$ ENABLE_JITPROFILING=1 perf record -o /tmp/perf.data --call-graph dwarf -k 1 ./julia
↪  /test/fastmath.jl
$ perf inject --jit --input /tmp/perf.data --output /tmp/perf-jit.data
$ perf report --call-graph -G -i /tmp/perf-jit.data
```

There are many more interesting things that you can measure about your program, to get a comprehensive list please read the Linux perf examples page.

Remember that perf saves for each execution a `perf.data` file that, even for small programs, can get quite large. Also the perf LLVM module saves temporarily debug objects in `~/.debug/jit`, remember to clean that folder frequently.

Chapter 34

Stack Traces

The StackTraces module provides simple stack traces that are both human readable and easy to use programmatically.

34.1 Viewing a stack trace

The primary function used to obtain a stack trace is `stacktrace`:

```
6-element Array{Base.StackTraces.StackFrame,1}:
 top-level scope
 eval at boot.jl:317 [inlined]
 eval(::Module, ::Expr) at REPL.jl:5
 eval_user_input(::Any, ::REPL.REPLBackend) at REPL.jl:85
 macro expansion at REPL.jl:116 [inlined]
 (::getfield(REPL, Symbol("##28#29")){REPL.REPLBackend})() at event.jl:92
```

Calling `stacktrace()` returns a vector of `StackTraces.StackFrame` s. For ease of use, the alias `StackTraces.StackTrace` can be used in place of `Vector{StackFrame}`. (Examples with `[...]` indicate that output may vary depending on how the code is run.)

```
julia> example() = stacktrace()
example (generic function with 1 method)

julia> example()
7-element Array{Base.StackTraces.StackFrame,1}:
 example() at REPL[1]:1
 top-level scope
 eval at boot.jl:317 [inlined]
[...]

julia> @noinline child() = stacktrace()
child (generic function with 1 method)

julia> @noinline parent() = child()
parent (generic function with 1 method)
```

```
julia> grandparent() = parent()
grandparent (generic function with 1 method)

julia> grandparent()
9-element Array{Base.StackTraces.StackFrame,1}:
 child() at REPL[3]:1
 parent() at REPL[4]:1
 grandparent() at REPL[5]:1
[...]
```

Note that when calling `stacktrace()` you'll typically see a frame with `eval at boot.jl`. When calling `stacktrace()` from the REPL you'll also have a few extra frames in the stack from `REPL.jl`, usually looking something like this:

```
julia> example() = stacktrace()
example (generic function with 1 method)

julia> example()
7-element Array{Base.StackTraces.StackFrame,1}:
 example() at REPL[1]:1
 top-level scope
 eval at boot.jl:317 [inlined]
 eval(::Module, ::Expr) at REPL.jl:5
 eval_user_input(::Any, ::REPL.REPLBackend) at REPL.jl:85
 macro expansion at REPL.jl:116 [inlined]
 (::getfield(REPL, Symbol("##28#29")){REPL.REPLBackend})() at event.jl:92
```

34.2 Extracting useful information

Each `StackTraces.StackFrame` contains the function name, file name, line number, lambda info, a flag indicating whether the frame has been inlined, a flag indicating whether it is a C function (by default C functions do not appear in the stack trace), and an integer representation of the pointer returned by `backtrace`:

```
julia> frame = stacktrace()[3]
eval(::Module, ::Expr) at REPL.jl:5

julia> frame.func
:eval

julia> frame.file
Symbol("~/julia/usr/share/julia/stdlib/v0.7/REPL/src/REPL.jl")

julia> frame.line
5

julia> frame.linfo
MethodInstance for eval(::Module, ::Expr)

julia> frame.inlined
false
```

```
julia> frame.from_c
false

julia> frame.pointer
0x00007f92d6293171
```

This makes stack trace information available programmatically for logging, error handling, and more.

34.3 Error handling

While having easy access to information about the current state of the callstack can be helpful in many places, the most obvious application is in error handling and debugging.

```
julia> @noinline bad_function() = undeclared_variable
bad_function (generic function with 1 method)

julia> @noinline example() = try
           bad_function()
       catch
           stacktrace()
       end
example (generic function with 1 method)

julia> example()
7-element Array{Base.StackTraces.StackFrame,1}:
 example() at REPL[2]:4
 top-level scope
 eval at boot.jl:317 [inlined]
[...]
```

You may notice that in the example above the first stack frame points at line 4, where `stacktrace` is called, rather than line 2, where *bad_function* is called, and `bad_function`'s frame is missing entirely. This is understandable, given that `stacktrace` is called from the context of the *catch*. While in this example it's fairly easy to find the actual source of the error, in complex cases tracking down the source of the error becomes nontrivial.

This can be remedied by passing the result of `catch_backtrace` to `stacktrace`. Instead of returning callstack information for the current context, `catch_backtrace` returns stack information for the context of the most recent exception:

```
julia> @noinline bad_function() = undeclared_variable
bad_function (generic function with 1 method)

julia> @noinline example() = try
           bad_function()
       catch
           stacktrace(catch_backtrace())
       end
example (generic function with 1 method)
```

```
julia> example()
8-element Array{Base.StackTraces.StackFrame,1}:
 bad_function() at REPL[1]:1
 example() at REPL[2]:2
[...]
```

Notice that the stack trace now indicates the appropriate line number and the missing frame.

```
julia> @noinline child() = error("Whoops!")
child (generic function with 1 method)

julia> @noinline parent() = child()
parent (generic function with 1 method)

julia> @noinline function grandparent()
           try
               parent()
           catch err
               println("ERROR: ", err.msg)
               stacktrace(catch_backtrace())
           end
       end
grandparent (generic function with 1 method)

julia> grandparent()
ERROR: Whoops!
10-element Array{Base.StackTraces.StackFrame,1}:
 error at error.jl:33 [inlined]
 child() at REPL[1]:1
 parent() at REPL[2]:1
 grandparent() at REPL[3]:3
[...]
```

34.4 Exception stacks and `current_exceptions`

Julia 1.1

Exception stacks requires at least Julia 1.1.

While handling an exception further exceptions may be thrown. It can be useful to inspect all these exceptions to identify the root cause of a problem. The julia runtime supports this by pushing each exception onto an internal *exception stack* as it occurs. When the code exits a `catch` normally, any exceptions which were pushed onto the stack in the associated `try` are considered to be successfully handled and are removed from the stack.

The stack of current exceptions can be accessed using the `current_exceptions` function. For example,

```
julia> try
           error("(A) The root cause")
       catch
           try
```

```
            error("(B) An exception while handling the exception")
        catch
            for (exc, bt) in current_exceptions()
                showerror(stdout, exc, bt)
                println(stdout)
            end
        end
    end
(A) The root cause
Stacktrace:
 [1] error(::String) at error.jl:33
 [2] top-level scope at REPL[7]:2
 [3] eval(::Module, ::Any) at boot.jl:319
 [4] eval_user_input(::Any, ::REPL.REPLBackend) at REPL.jl:85
 [5] macro expansion at REPL.jl:117 [inlined]
 [6] (::getfield(REPL, Symbol("##26#27")){REPL.REPLBackend})() at task.jl:259
(B) An exception while handling the exception
Stacktrace:
 [1] error(::String) at error.jl:33
 [2] top-level scope at REPL[7]:5
 [3] eval(::Module, ::Any) at boot.jl:319
 [4] eval_user_input(::Any, ::REPL.REPLBackend) at REPL.jl:85
 [5] macro expansion at REPL.jl:117 [inlined]
 [6] (::getfield(REPL, Symbol("##26#27")){REPL.REPLBackend})() at task.jl:259
```

In this example the root cause exception (A) is first on the stack, with a further exception (B) following it. After exiting both catch blocks normally (i.e., without throwing a further exception) all exceptions are removed from the stack and are no longer accessible.

The exception stack is stored on the `Task` where the exceptions occurred. When a task fails with uncaught exceptions, `current_exceptions(task)` may be used to inspect the exception stack for that task.

34.5 Comparison with `backtrace`

A call to `backtrace` returns a vector of `Union{Ptr{Nothing}, Base.InterpreterIP}`, which may then be passed into `stacktrace` for translation:

```
julia> trace = backtrace()
18-element Array{Union{Ptr{Nothing}, Base.InterpreterIP},1}:
 Ptr{Nothing} @0x00007fd8734c6209
 Ptr{Nothing} @0x00007fd87362b342
 Ptr{Nothing} @0x00007fd87362c136
 Ptr{Nothing} @0x00007fd87362c986
 Ptr{Nothing} @0x00007fd87362d089
 Base.InterpreterIP(CodeInfo(:(begin
      Core.SSAValue(0) = backtrace()
      trace = Core.SSAValue(0)
      return Core.SSAValue(0)
  end)), 0x0000000000000000)
 Ptr{Nothing} @0x00007fd87362e4cf
[...]
```

```
julia> stacktrace(trace)
6-element Array{Base.StackTraces.StackFrame,1}:
 top-level scope
 eval at boot.jl:317 [inlined]
 eval(::Module, ::Expr) at REPL.jl:5
 eval_user_input(::Any, ::REPL.REPLBackend) at REPL.jl:85
 macro expansion at REPL.jl:116 [inlined]
 (::getfield(REPL, Symbol("##28#29")){REPL.REPLBackend})() at event.jl:92
```

Notice that the vector returned by `backtrace` had 18 elements, while the vector returned by `stacktrace` only has 6. This is because, by default, `stacktrace` removes any lower-level C functions from the stack. If you want to include stack frames from C calls, you can do it like this:

```
julia> stacktrace(trace, true)
21-element Array{Base.StackTraces.StackFrame,1}:
 jl_apply_generic at gf.c:2167
 do_call at interpreter.c:324
 eval_value at interpreter.c:416
 eval_body at interpreter.c:559
 jl_interpret_toplevel_thunk_callback at interpreter.c:798
 top-level scope
 jl_interpret_toplevel_thunk at interpreter.c:807
 jl_toplevel_eval_flex at toplevel.c:856
 jl_toplevel_eval_in at builtins.c:624
 eval at boot.jl:317 [inlined]
 eval(::Module, ::Expr) at REPL.jl:5
 jl_apply_generic at gf.c:2167
 eval_user_input(::Any, ::REPL.REPLBackend) at REPL.jl:85
 jl_apply_generic at gf.c:2167
 macro expansion at REPL.jl:116 [inlined]
 (::getfield(REPL, Symbol("##28#29")){REPL.REPLBackend})() at event.jl:92
 jl_fptr_trampoline at gf.c:1838
 jl_apply_generic at gf.c:2167
 jl_apply at julia.h:1540 [inlined]
 start_task at task.c:268
 ip:0xffffffffffffffff
```

Individual pointers returned by `backtrace` can be translated into `StackTraces.StackFrame` s by passing them into `StackTraces.lookup`:

```
julia> pointer = backtrace()[1];

julia> frame = StackTraces.lookup(pointer)
1-element Array{Base.StackTraces.StackFrame,1}:
 jl_apply_generic at gf.c:2167

julia> println("The top frame is from $(frame[1].func)!")
The top frame is from jl_apply_generic!
```

Chapter 35

Memory Management and Garbage Collection

Julia uses automatic memory management through its built-in garbage collector (GC). This section provides an overview of how Julia manages memory and how you can configure and optimize memory usage for your applications.

35.1 Garbage Collection Overview

Julia features a garbage collector with the following characteristics:

- **Non-moving**: Objects are not relocated in memory during garbage collection
- **Generational**: Younger objects are collected more frequently than older ones
- **Parallel and partially concurrent**: The GC can use multiple threads and run concurrently with your program
- **Mostly precise**: The GC accurately identifies object references for pure Julia code, and it provides conservative scanning APIs for users calling Julia from C

The garbage collector automatically reclaims memory used by objects that are no longer reachable from your program, freeing you from manual memory management in most cases.

35.2 Memory Architecture

Julia uses a two-tier allocation strategy:

- **Small objects** (currently ≤ 2032 bytes but may change): Allocated using a fast per-thread pool allocator
- **Large objects** : Allocated directly through the system's `malloc`

This hybrid approach optimizes for both allocation speed and memory efficiency, with the pool allocator providing fast allocation for the many small objects typical in Julia programs.

35.3 System Memory Requirements

Swap Space

Julia's garbage collector is designed with the expectation that your system has adequate swap space configured. The GC uses heuristics that assume it can allocate memory beyond physical RAM when needed, relying on the operating system's virtual memory management.

If your system has limited or no swap space, you may experience out-of-memory errors during garbage collection. In such cases, you can use the `--heap-size-hint` option to limit Julia's memory usage.

Memory Hints

You can provide a hint to Julia about the maximum amount of memory to use:

```
julia --heap-size-hint=4G  # To set the hint to ~4GB
julia --heap-size-hint=50% # or to 50% of physical memory
```

The `--heap-size-hint` option tells the garbage collector to trigger collection more aggressively when approaching the specified limit. This is particularly useful in:

- Containers with memory limits
- Systems without swap space
- Shared systems where you want to limit Julia's memory footprint

You can also set this via the `JULIA_HEAP_SIZE_HINT` environment variable:

```
export JULIA_HEAP_SIZE_HINT=2G
julia
```

35.4 Multithreaded Garbage Collection

Julia's garbage collector can leverage multiple threads to improve performance on multi-core systems.

GC Thread Configuration

By default, Julia uses multiple threads for garbage collection:

- **Mark threads**: Used during the mark phase to trace object references (default: 1, which is shared with the compute thread if there is only one, otherwise half the number of compute threads)
- **Sweep threads**: Used for concurrent sweeping of freed memory (default: 0, disabled)

You can configure GC threading using:

```
julia --gcthreads=4,1  # 4 mark threads, 1 sweep thread
julia --gcthreads=8    # 8 mark threads, 0 sweep threads
```

Or via environment variable:

```
export JULIA_NUM_GC_THREADS=4,1
julia
```

Recommendations

For compute-intensive workloads:

- Use multiple mark threads (the default configuration is usually appropriate)
- Consider enabling concurrent sweeping with 1 sweep thread for allocation-heavy workloads

For memory-intensive workloads:

- Enable concurrent sweeping to reduce GC pauses
- Monitor GC time using `@time` and adjust thread counts accordingly

35.5 Monitoring and Debugging

Basic Memory Monitoring

Use the `@time` macro to see memory allocation and GC overhead:

```
julia> @time some_computation()
  2.123456 seconds (1.50 M allocations: 58.725 MiB, 17.17% gc time)
```

GC Logging

Enable detailed GC logging to understand collection patterns:

```
julia> GC.enable_logging(true)
julia> # Run your code
julia> GC.enable_logging(false)
```

This logs each garbage collection event with timing and memory statistics.

Manual GC Control

While generally not recommended, you can manually trigger garbage collection:

```
GC.gc()          # Force a garbage collection
GC.enable(false) # Disable automatic GC (use with caution!)
GC.enable(true)  # Re-enable automatic GC
```

Warning: Disabling GC can lead to memory exhaustion. Only use this for specific performance measurements or debugging.

35.6 Performance Considerations

Reducing Allocations

The best way to minimize GC impact is to reduce unnecessary allocations:

- Use in-place operations when possible (e.g., `x .+= y` instead of `x = x + y`)
- Pre-allocate arrays and reuse them
- Avoid creating temporary objects in tight loops
- Consider using `StaticArrays.jl` for small, fixed-size arrays

Memory-Efficient Patterns

- Avoid global variables that change type
- Use `const` for global constants

Profiling Memory Usage

For detailed guidance on profiling memory allocations and identifying performance bottlenecks, see the Profiling section.

35.7 Advanced Configuration

Integration with System Memory Management

Julia works best when:

- The system has adequate swap space (recommended: 2x physical RAM)
- Virtual memory is properly configured
- Other processes leave sufficient memory available
- Container memory limits are set appropriately with `--heap-size-hint`

35.8 Troubleshooting Memory Issues

High GC Overhead

If garbage collection is taking too much time:

1. **Reduce allocation rate**: Focus on algorithmic improvements
2. **Adjust GC threads**: Experiment with different `--gcthreads` settings
3. **Use concurrent sweeping**: Enable background sweeping with `--gcthreads=N,1`
4. **Profile memory patterns**: Identify allocation hotspots and optimize them

Memory Leaks

While Julia's GC prevents most memory leaks, issues can still occur:

- **Global references**: Avoid holding references to large objects in global variables
- **Closures**: Be careful with closures that capture large amounts of data
- **C interop**: Ensure proper cleanup when interfacing with C libraries

For more detailed information about Julia's garbage collector internals, see the Garbage Collection section in the Developer Documentation.

Chapter 36

Performance Tips

In the following sections, we briefly go through a few techniques that can help make your Julia code run as fast as possible.

36.1 Table of contents

36.2 General advice

Performance critical code should be inside a function

Any code that is performance critical should be inside a function. Code inside functions tends to run much faster than top level code, due to how Julia's compiler works.

The use of functions is not only important for performance: functions are more reusable and testable, and clarify what steps are being done and what their inputs and outputs are, Write functions, not just scripts is also a recommendation of Julia's Styleguide.

The functions should take arguments, instead of operating directly on global variables, see the next point.

Avoid untyped global variables

The value of an untyped global variable might change at any point, possibly leading to a change of its type. This makes it difficult for the compiler to optimize code using global variables. This also applies to type-valued variables, i.e. type aliases on the global level. Variables should be local, or passed as arguments to functions, whenever possible.

We find that global names are frequently constants, and declaring them as such greatly improves performance:

```
const DEFAULT_VAL = 0
```

If a non-constant global is known to always be of the same type, the type should be annotated; `const` globals need not be annotated because their type is inferred from their initialization value.

Uses of untyped globals can be optimized by annotating their types at the point of use:

```
global x = rand(1000)

function loop_over_global()
    s = 0.0
    for i in x::Vector{Float64}
        s += i
    end
    return s
end
```

Passing arguments to functions is better style. It leads to more reusable code and clarifies what the inputs and outputs are.

Note

All code in the REPL is evaluated in global scope, so a variable defined and assigned at top level will be a **global** variable. Variables defined at top level scope inside modules are also global.

In the following REPL session:

```
julia> x = 1.0
1.0
```

is equivalent to:

```
julia> global x = 1.0
1.0
```

so all the performance issues discussed previously apply.

Measure performance with `@time` and pay attention to memory allocation

A useful tool for measuring performance is the `@time` macro. We here repeat the example with the global variable above, but this time with the type annotation removed:

```
julia> x = rand(1000);

julia> function sum_global()
           s = 0.0
           for i in x
               s += i
           end
           return s
       end;
```

```
julia> @time sum_global()
  0.011539 seconds (9.08 k allocations: 373.386 KiB, 98.69% compilation time)
523.0007221951678

julia> @time sum_global()
  0.000091 seconds (3.49 k allocations: 70.156 KiB)
523.0007221951678
```

On the first call (`@time sum_global()`) the function gets compiled. (If you've not yet used `@time` in this session, it will also compile functions needed for timing.) You should not take the results of this run seriously. For the second run, note that in addition to reporting the time, it also indicated that a significant amount of memory was allocated. We are here just computing a sum over all elements in a vector of 64-bit floats so there should be no need to allocate (heap) memory.

We should clarify that what `@time` reports is specifically *heap* allocations, which are typically needed for either mutable objects or for creating/growing variable-sized containers (such as `Array` or `Dict`, strings, or "type-unstable" objects whose type is only known at runtime). Allocating (or deallocating) such blocks of memory may require an expensive function call to libc (e.g. via `malloc` in C), and they must be tracked for garbage collection. In contrast, immutable values like numbers (except bignums), tuples, and immutable `struct`s can be stored much more cheaply, e.g. in stack or CPU-register memory, so one doesn't typically worry about the performance cost of "allocating" them.

Unexpected memory allocation is almost always a sign of some problem with your code, usually a problem with type-stability or creating many small temporary arrays. Consequently, in addition to the allocation itself, it's very likely that the code generated for your function is far from optimal. Take such indications seriously and follow the advice below.

For more information about memory management and garbage collection in Julia, see Memory Management and Garbage Collection.

In this particular case, the memory allocation is due to the usage of a type-unstable global variable `x`, so if we instead pass `x` as an argument to the function it no longer allocates memory (the remaining allocation reported below is due to running the `@time` macro in global scope) and is significantly faster after the first call:

```
julia> x = rand(1000);

julia> function sum_arg(x)
           s = 0.0
           for i in x
               s += i
           end
           return s
       end;

julia> @time sum_arg(x)
  0.007551 seconds (3.98 k allocations: 200.548 KiB, 99.77% compilation time)
523.0007221951678

julia> @time sum_arg(x)
  0.000006 seconds (1 allocation: 16 bytes)
523.0007221951678
```

The 1 allocation seen is from running the `@time` macro itself in global scope. If we instead run the timing in a function, we can see that indeed no allocations are performed:

```
julia> time_sum(x) = @time sum_arg(x);

julia> time_sum(x)
  0.000002 seconds
523.0007221951678
```

In some situations, your function may need to allocate memory as part of its operation, and this can complicate the simple picture above. In such cases, consider using one of the tools below to diagnose problems, or write a version of your function that separates allocation from its algorithmic aspects (see Pre-allocate outputs).

Note

For more serious benchmarking, consider the BenchmarkTools.jl package which among other things evaluates the function multiple times in order to reduce noise.

Break functions into multiple definitions

Writing a function as many small definitions allows the compiler to directly call the most applicable code, or even inline it.

Here is an example of a "compound function" that should really be written as multiple definitions:

```
using LinearAlgebra

function mynorm(A)
    if isa(A, Vector)
        return sqrt(real(dot(A,A)))
    elseif isa(A, Matrix)
        return maximum(svdvals(A))
    else
        error("mynorm: invalid argument")
    end
end
```

This can be written more concisely and efficiently as:

```
mynorm(x::Vector) = sqrt(real(dot(x, x)))
mynorm(A::Matrix) = maximum(svdvals(A))
```

It should however be noted that the compiler is quite efficient at optimizing away the dead branches in code written as the `mynorm` example.

Tools

Julia and its package ecosystem includes tools that may help you diagnose problems and improve the performance of your code:

- Profiling allows you to measure the performance of your running code and identify lines that serve as bottlenecks. For complex projects, the ProfileView package can help you visualize your profiling results.
- The JET package can help you find common performance problems in your code.
- Unexpectedly-large memory allocations–as reported by `@time`, `@allocated`, or the profiler (through calls to the garbage-collection routines)–hint that there might be issues with your code. If you don't see another reason for the allocations, suspect a type problem. You can also start Julia with the `--track-allocation=user` option and examine the resulting `*.mem` files to see information about where those allocations occur. See Memory allocation analysis.
- `@code_warntype` generates a representation of your code that can be helpful in finding expressions that result in type uncertainty. See `@code_warntype` below.

36.3 Type inference

In many languages with optional type declarations, adding declarations is the principal way to make code run faster. This is *not* the case in Julia. In Julia, the compiler generally knows the types of all function arguments, local variables, and expressions. However, there are a few specific instances where declarations are helpful.

Avoid containers with abstract type parameters

When working with parameterized types, including arrays, it is best to avoid parameterizing with abstract types where possible.

Consider the following:

```
julia> a = Real[]
Real[]

julia> push!(a, 1); push!(a, 2.0); push!(a, π)
3-element Vector{Real}:
 1
 2.0
 π = 3.1415926535897...
```

Because a is an array of abstract type `Real`, it must be able to hold any `Real` value. Since `Real` objects can be of arbitrary size and structure, a must be represented as an array of pointers to individually allocated `Real` objects. However, if we instead only allow numbers of the same type, e.g. `Float64`, to be stored in a these can be stored more efficiently:

```
julia> a = Float64[]
Float64[]
```

```
julia> push!(a, 1); push!(a, 2.0); push!(a,  π)
3-element Vector{Float64}:
 1.0
 2.0
 3.141592653589793
```

Assigning numbers into a will now convert them to `Float64` and a will be stored as a contiguous block of 64-bit floating-point values that can be manipulated efficiently.

If you cannot avoid containers with abstract value types, it is sometimes better to parametrize with `Any` to avoid runtime type checking. E.g. `IdDict{Any, Any}` performs better than `IdDict{Type, Vector}`

See also the discussion under Parametric Types.

Avoid fields with abstract type

Types can be declared without specifying the types of their fields:

```
julia> struct MyAmbiguousType
           a
       end
```

This allows a to be of any type. This can often be useful, but it does have a downside: for objects of type `MyAmbiguousType`, the compiler will not be able to generate high-performance code. The reason is that the compiler uses the types of objects, not their values, to determine how to build code. Unfortunately, very little can be inferred about an object of type `MyAmbiguousType`:

```
julia> b = MyAmbiguousType("Hello")
MyAmbiguousType("Hello")

julia> c = MyAmbiguousType(17)
MyAmbiguousType(17)

julia> typeof(b)
MyAmbiguousType

julia> typeof(c)
MyAmbiguousType
```

The values of b and c have the same type, yet their underlying representation of data in memory is very different. Even if you stored just numeric values in field a, the fact that the memory representation of a `UInt8` differs from a `Float64` also means that the CPU needs to handle them using two different kinds of instructions. Since the required information is not available in the type, such decisions have to be made at run-time. This slows performance.

You can do better by declaring the type of a. Here, we are focused on the case where a might be any one of several types, in which case the natural solution is to use parameters. For example:

```
julia> mutable struct MyType{T<:AbstractFloat}
           a::T
       end
```

This is a better choice than

```
julia> mutable struct MyStillAmbiguousType
           a::AbstractFloat
       end
```

because the first version specifies the type of a from the type of the wrapper object. For example:

```
julia> m = MyType(3.2)
MyType{Float64}(3.2)

julia> t = MyStillAmbiguousType(3.2)
MyStillAmbiguousType(3.2)

julia> typeof(m)
MyType{Float64}

julia> typeof(t)
MyStillAmbiguousType
```

The type of field a can be readily determined from the type of `m`, but not from the type of `t`. Indeed, in `t` it's possible to change the type of the field a:

```
julia> typeof(t.a)
Float64

julia> t.a = 4.5f0
4.5f0

julia> typeof(t.a)
Float32
```

In contrast, once `m` is constructed, the type of `m.a` cannot change:

```
julia> m.a = 4.5f0
4.5f0

julia> typeof(m.a)
Float64
```

The fact that the type of `m.a` is known from `m`'s type—coupled with the fact that its type cannot change mid-function—allows the compiler to generate highly-optimized code for objects like `m` but not for objects like `t`.

Of course, all of this is true only if we construct `m` with a concrete type. We can break this by explicitly constructing it with an abstract type:

```
julia> m = MyType{AbstractFloat}(3.2)
MyType{AbstractFloat}(3.2)

julia> typeof(m.a)
Float64

julia> m.a = 4.5f0
4.5f0

julia> typeof(m.a)
Float32
```

For all practical purposes, such objects behave identically to those of `MyStillAmbiguousType`.

It's quite instructive to compare the sheer amount of code generated for a simple function

```
func(m::MyType) = m.a+1
```

using

```
code_llvm(func, Tuple{MyType{Float64}})
code_llvm(func, Tuple{MyType{AbstractFloat}})
```

For reasons of length the results are not shown here, but you may wish to try this yourself. Because the type is fully-specified in the first case, the compiler doesn't need to generate any code to resolve the type at run-time. This results in shorter and faster code.

One should also keep in mind that not-fully-parameterized types behave like abstract types. For example, even though a fully specified `Array{T,n}` is concrete, `Array` itself with no parameters given is not concrete:

```
julia> !isconcretetype(Array), !isabstracttype(Array), isstructtype(Array),
↪  !isconcretetype(Array{Int}), isconcretetype(Array{Int,1})
(true, true, true, true, true)
```

In this case, it would be better to avoid declaring `MyType` with a field `a::Array` and instead declare the field as `a::Array{T,N}` or as `a::A`, where `{T,N}` or `A` are parameters of `MyType`.

The previous advice is especially useful when the fields of a struct are meant to be functions, or more generally callable objects. It is very tempting to define a struct as follows:

```
struct MyCallableWrapper
    f::Function
end
```

But since `Function` is an abstract type, every call to `wrapper.f` will require dynamic dispatch, due to the type instability of accessing the field `f`. Instead, you should write something like:

```
struct MyCallableWrapper{F}
    f::F
end
```

which has nearly identical behavior but will be much faster (because the type instability is eliminated). Note that we do not impose F<:Function: this means callable objects which do not subtype Function are also allowed for the field f.

Avoid fields with abstract containers

The same best practices also work for container types:

```
julia> struct MySimpleContainer{A<:AbstractVector}
           a::A
       end

julia> struct MyAmbiguousContainer{T}
           a::AbstractVector{T}
       end

julia> struct MyAlsoAmbiguousContainer
           a::Array
       end
```

For example:

```
julia> c = MySimpleContainer(1:3);

julia> typeof(c)
MySimpleContainer{UnitRange{Int64}}

julia> c = MySimpleContainer([1:3;]);

julia> typeof(c)
MySimpleContainer{Vector{Int64}}

julia> b = MyAmbiguousContainer(1:3);

julia> typeof(b)
MyAmbiguousContainer{Int64}

julia> b = MyAmbiguousContainer([1:3;]);

julia> typeof(b)
MyAmbiguousContainer{Int64}

julia> d = MyAlsoAmbiguousContainer(1:3);

julia> typeof(d), typeof(d.a)
(MyAlsoAmbiguousContainer, Vector{Int64})

julia> d = MyAlsoAmbiguousContainer(1:1.0:3);

julia> typeof(d), typeof(d.a)
(MyAlsoAmbiguousContainer, Vector{Float64})
```

For `MySimpleContainer`, the object is fully-specified by its type and parameters, so the compiler can generate optimized functions. In most instances, this will probably suffice.

While the compiler can now do its job perfectly well, there are cases where *you* might wish that your code could do different things depending on the *element type* of a. Usually the best way to achieve this is to wrap your specific operation (here, `foo`) in a separate function:

```
julia> function sumfoo(c::MySimpleContainer)
           s = 0
           for x in c.a
               s += foo(x)
           end
           s
       end
sumfoo (generic function with 1 method)

julia> foo(x::Integer) = x
foo (generic function with 1 method)

julia> foo(x::AbstractFloat) = round(x)
foo (generic function with 2 methods)
```

This keeps things simple, while allowing the compiler to generate optimized code in all cases.

However, there are cases where you may need to declare different versions of the outer function for different element types or types of the `AbstractVector` of the field a in `MySimpleContainer`. You could do it like this:

```
julia> function myfunc(c::MySimpleContainer{<:AbstractArray{<:Integer}})
           return c.a[1]+1
       end
myfunc (generic function with 1 method)

julia> function myfunc(c::MySimpleContainer{<:AbstractArray{<:AbstractFloat}})
           return c.a[1]+2
       end
myfunc (generic function with 2 methods)

julia> function myfunc(c::MySimpleContainer{Vector{T}}) where T <: Integer
           return c.a[1]+3
       end
myfunc (generic function with 3 methods)
```

```
julia> myfunc(MySimpleContainer(1:3))
2

julia> myfunc(MySimpleContainer(1.0:3))
3.0

julia> myfunc(MySimpleContainer([1:3;]))
4
```

Annotate values taken from untyped locations

It is often convenient to work with data structures that may contain values of any type (arrays of type `Array{Any}`). But, if you're using one of these structures and happen to know the type of an element, it helps to share this knowledge with the compiler:

```
function foo(a::Array{Any,1})
    x = a[1]::Int32
    b = x+1
    ...
end
```

Here, we happened to know that the first element of a would be an `Int32`. Making an annotation like this has the added benefit that it will raise a run-time error if the value is not of the expected type, potentially catching certain bugs earlier.

In the case that the type of `a[1]` is not known precisely, `x` can be declared via `x = convert(Int32, a[1])::Int32`. The use of the `convert` function allows `a[1]` to be any object convertible to an `Int32` (such as `UInt8`), thus increasing the genericity of the code by loosening the type requirement. Notice that `convert` itself needs a type annotation in this context in order to achieve type stability. This is because the compiler cannot deduce the type of the return value of a function, even `convert`, unless the types of all the function's arguments are known.

Type annotation will not enhance (and can actually hinder) performance if the type is abstract, or constructed at run-time. This is because the compiler cannot use the annotation to specialize the subsequent code, and the type-check itself takes time. For example, in the code:

```
function nr(a, prec)
    ctype = prec == 32 ? Float32 : Float64
    b = Complex{ctype}(a)
    c = (b + 1.0f0)::Complex{ctype}
    abs(c)
end
```

the annotation of c harms performance. To write performant code involving types constructed at run-time, use the function-barrier technique discussed below, and ensure that the constructed type appears among the argument types of the kernel function so that the kernel operations are properly specialized by the compiler. For example, in the above snippet, as soon as b is constructed, it can be passed to another function k, the kernel. If, for example, function k declares b as an argument of type `Complex{T}`, where T is a type parameter, then a type annotation appearing in an assignment statement within k of the form:

```
c = (b + 1.0f0)::Complex{T}
```

does not hinder performance (but does not help either) since the compiler can determine the type of c at the time k is compiled.

Be aware of when Julia avoids specializing

As a heuristic, Julia avoids automatically specializing on argument type parameters in three specific cases: `Type`, `Function`, and `Vararg`. Julia will always specialize when the argument is used within the method, but not if the argument is just passed through to another function. This usually has no performance impact at runtime and improves compiler performance. If you find it does have a performance impact at runtime in your case, you can trigger specialization by adding a type parameter to the method declaration. Here are some examples:

This will not specialize:

```
function f_type(t)  # or t::Type
    x = ones(t, 10)
    return sum(map(sin, x))
end
```

but this will:

```
function g_type(t::Type{T}) where T
    x = ones(T, 10)
    return sum(map(sin, x))
end
```

These will not specialize:

```
f_func(f, num) = ntuple(f, div(num, 2))
g_func(g::Function, num) = ntuple(g, div(num, 2))
```

but this will:

```
h_func(h::H, num) where {H} = ntuple(h, div(num, 2))
```

This will not specialize:

```
f_vararg(x::Int...) = tuple(x...)
```

but this will:

```
g_vararg(x::Vararg{Int, N}) where {N} = tuple(x...)
```

One only needs to introduce a single type parameter to force specialization, even if the other types are unconstrained. For example, this will also specialize, and is useful when the arguments are not all of the same type:

```
h_vararg(x::Vararg{Any, N}) where {N} = tuple(x...)
```

Note that `@code_typed` and friends will always show you specialized code, even if Julia would not normally specialize that method call. You need to check the method internals if you want to see whether specializations are generated when argument types are changed, i.e., if `Base.specializations(@which f(...))` contains specializations for the argument in question.

Write "type-stable" functions

When possible, it helps to ensure that a function always returns a value of the same type. Consider the following definition:

```
pos(x) = x < 0 ? 0 : x
```

Although this seems innocent enough, the problem is that `0` is an integer (of type `Int`) and `x` might be of any type. Thus, depending on the value of `x`, this function might return a value of either of two types. This behavior is allowed, and may be desirable in some cases. But it can easily be fixed as follows:

```
pos(x) = x < 0 ? zero(x) : x
```

There is also a `oneunit` function, and a more general `oftype(x, y)` function, which returns `y` converted to the type of `x`.

Avoid changing the type of a variable

An analogous "type-stability" problem exists for variables used repeatedly within a function:

```
function foo()
    x = 1
    for i = 1:10
        x /= rand()
    end
    return x
end
```

Local variable `x` starts as an integer, and after one loop iteration becomes a floating-point number (the result of `/` operator). This makes it more difficult for the compiler to optimize the body of the loop. There are several possible fixes:

- Initialize `x` with `x = 1.0`
- Declare the type of `x` explicitly as `x::Float64 = 1`
- Use an explicit conversion by `x = oneunit(Float64)`
- Initialize with the first loop iteration, to `x = 1 / rand()`, then loop `for i = 2:10`

Separate kernel functions (aka, function barriers)

Many functions follow a pattern of performing some set-up work, and then running many iterations to perform a core computation. Where possible, it is a good idea to put these core computations in separate functions. For example, the following contrived function returns an array of a randomly-chosen type:

```
julia> function strange_twos(n)
           a = Vector{rand(Bool) ? Int64 : Float64}(undef, n)
           for i = 1:n
               a[i] = 2
           end
           return a
       end;

julia> strange_twos(3)
3-element Vector{Int64}:
 2
 2
 2
```

This should be written as:

```
julia> function fill_twos!(a)
           for i = eachindex(a)
               a[i] = 2
           end
       end;

julia> function strange_twos(n)
           a = Vector{rand(Bool) ? Int64 : Float64}(undef, n)
           fill_twos!(a)
           return a
       end;

julia> strange_twos(3)
3-element Vector{Int64}:
 2
 2
 2
```

Julia's compiler specializes code for argument types at function boundaries, so in the original implementation it does not know the type of a during the loop (since it is chosen randomly). Therefore the second version is generally faster since the inner loop can be recompiled as part of `fill_twos!` for different types of a.

The second form is also often better style and can lead to more code reuse.

This pattern is used in several places in Julia Base. For example, see `vcat` and `hcat` in `abstractarray.jl`, or the `fill!` function, which we could have used instead of writing our own `fill_twos!`.

Functions like `strange_twos` occur when dealing with data of uncertain type, for example data loaded from an input file that might contain either integers, floats, strings, or something else.

`@code_warntype`

The macro `@code_warntype` (or its function variant `code_warntype`) can sometimes be helpful in diagnosing type-related problems. Here's an example:

```
julia> @noinline pos(x) = x < 0 ? 0 : x;

julia> function f(x)
           y = pos(x)
           return sin(y*x + 1)
       end;

julia> @code_warntype f(3.2)
MethodInstance for f(::Float64)
  from f(x) @ Main REPL[9]:1
Arguments
  #self#::Core.Const(f)
  x::Float64
Locals
  y::Union{Float64, Int64}
Body::Float64
1 —      (y = Main.pos(x))
│   %2 = (y * x)::Float64
│   %3 = (%2 + 1)::Float64
│   %4 = Main.sin(%3)::Float64
└──      return %4
```

Interpreting the output of `@code_warntype`, like that of its cousins `@code_lowered`, `@code_typed`, `@code_llvm`, and `@code_native`, takes a little practice. Your code is being presented in form that has been heavily digested on its way to generating compiled machine code. Most of the expressions are annotated by a type, indicated by the `::T` (where T might be `Float64`, for example). The most important characteristic of `@code_warntype` is that non-concrete types are displayed in red; since this document is written in Markdown, which has no color, in this document, red text is denoted by uppercase.

At the top, the inferred return type of the function is shown as `Body::Float64`. The next lines represent the body of `f` in Julia's SSA IR form. The numbered boxes are labels and represent targets for jumps (via `goto`) in your code. Looking at the body, you can see that the first thing that happens is that `pos` is called and the return value has been inferred as the `Union` type `Union{Float64, Int64}` shown in uppercase since it is a non-concrete type. This means that we cannot know the exact return type of `pos` based on the input types. However, the result of `y*x`is a `Float64` no matter if `y` is a `Float64` or `Int64` The net result is that `f(x::Float64)` will not be type-unstable in its output, even if some of the intermediate computations are type-unstable.

How you use this information is up to you. Obviously, it would be far and away best to fix `pos` to be type-stable: if you did so, all of the variables in `f` would be concrete, and its performance would be optimal. However, there are circumstances where this kind of *ephemeral* type instability might not matter too much: for example, if `pos` is never used in isolation, the fact that `f`'s output is type-stable (for `Float64` inputs) will shield later code from the propagating effects of type instability. This is particularly relevant in cases where fixing the type instability is difficult or impossible. In such cases, the tips above (e.g., adding type annotations and/or breaking up functions) are your best tools to contain the "damage" from type instability. Also, note that even Julia Base has functions that are type unstable. For example, the function `findfirst` returns the index into an array where a key is found, or `nothing` if it is not found, a clear type instability. In order to make it easier to find the type instabilities that are likely to be important, `Union`s containing either `missing` or `nothing` are color highlighted in yellow, instead of red.

The following examples may help you interpret expressions marked as containing non-concrete types:

- Function body starting with `Body::Union{T1,T2})`

  - Interpretation: function with unstable return type
  - Suggestion: make the return value type-stable, even if you have to annotate it

- `invoke Main.g(%%x::Int64)::Union{Float64, Int64}`
  - Interpretation: call to a type-unstable function `g`.
  - Suggestion: fix the function, or if necessary annotate the return value

- `invoke Base.getindex(%%x::Array{Any,1}, 1::Int64)::Any`
  - Interpretation: accessing elements of poorly-typed arrays
  - Suggestion: use arrays with better-defined types, or if necessary annotate the type of individual element accesses

- `Base.getfield(%%x, :(:data))::Array{Float64,N} where N`
  - Interpretation: getting a field that is of non-concrete type. In this case, the type of `x`, say `ArrayContainer`, had a field `data::Array{T}`. But `Array` needs the dimension `N`, too, to be a concrete type.
  - Suggestion: use concrete types like `Array{T,3}` or `Array{T,N}`, where `N` is now a parameter of `ArrayContainer`

Performance of captured variable

Consider the following example that defines an inner function:

```
function abmult(r::Int)
    if r < 0
        r = -r
    end
    f = x -> x * r
    return f
end
```

Function `abmult` returns a function `f` that multiplies its argument by the absolute value of `r`. The inner function assigned to `f` is called a "closure". Inner functions are also used by the language for do-blocks and for generator expressions.

This style of code presents performance challenges for the language. The parser, when translating it into lower-level instructions, substantially reorganizes the above code by extracting the inner function to a separate code block. "Captured" variables such as `r` that are shared by inner functions and their enclosing scope are also extracted into a heap-allocated "box" accessible to both inner and outer functions because the language specifies that `r` in the inner scope must be identical to `r` in the outer scope even after the outer scope (or another inner function) modifies `r`.

The discussion in the preceding paragraph referred to the "parser", that is, the phase of compilation that takes place when the module containing `abmult` is first loaded, as opposed to the later phase when it is first invoked. The parser does not "know" that `Int` is a fixed type, or that the statement `r = -r` transforms an `Int` to another `Int`. The magic of type inference takes place in the later phase of compilation.

Thus, the parser does not know that `r` has a fixed type (`Int`). Nor that `r` does not change value once the inner function is created (so that the box is unneeded). Therefore, the parser emits code for box that holds an object with an abstract type such as `Any`, which requires run-time type dispatch for each occurrence of `r`. This can be verified by applying `@code_warntype` to the above function. Both the boxing and the run-time type dispatch can cause loss of performance.

If captured variables are used in a performance-critical section of the code, then the following tips help ensure that their use is performant. First, if it is known that a captured variable does not change its type, then this can be declared explicitly with a type annotation (on the variable, not the right-hand side):

```
function abmult2(r0::Int)
    r::Int = r0
    if r < 0
        r = -r
    end
    f = x -> x * r
    return f
end
```

The type annotation partially recovers lost performance due to capturing because the parser can associate a concrete type to the object in the box. Going further, if the captured variable does not need to be boxed at all (because it will not be reassigned after the closure is created), this can be indicated with `let` blocks as follows.

```
function abmult3(r::Int)
    if r < 0
        r = -r
    end
    f = let r = r
            x -> x * r
    end
    return f
end
```

The `let` block creates a new variable `r` whose scope is only the inner function. The second technique recovers full language performance in the presence of captured variables. Note that this is a rapidly evolving aspect of the compiler, and it is likely that future releases will not require this degree of programmer annotation to attain performance. In the mean time, some user-contributed packages like FastClosures automate the insertion of `let` statements as in `abmult3`.

Use `@__FUNCTION__` for recursive closures

For recursive closures specifically, the `@__FUNCTION__` macro can avoid both type instability and boxing.

First, let's see the unoptimized version:

```
function make_fib_unoptimized()
    fib(n) = n <= 1 ? 1 : fib(n - 1) + fib(n - 2)  # fib is boxed
    return fib
end
```

The `fib` function is boxed, meaning the return type is inferred as `Any`:

```
@code_warntype make_fib_unoptimized()
```

Now, to eliminate this type instability, we can instead use `@__FUNCTION__` to refer to the concrete function object:

```
function make_fib_optimized()
    fib(n) = n <= 1 ? 1 : (@__FUNCTION__)(n - 1) + (@__FUNCTION__)(n - 2)
    return fib
end
```

This gives us a concrete return type:

```
@code_warntype make_fib_optimized()
```

Types with values-as-parameters

Let's say you want to create an `N`-dimensional array that has size 3 along each axis. Such arrays can be created like this:

```
julia> A = fill(5.0, (3, 3))
3×3 Matrix{Float64}:
 5.0  5.0  5.0
 5.0  5.0  5.0
 5.0  5.0  5.0
```

This approach works very well: the compiler can figure out that `A` is an `Array{Float64,2}` because it knows the type of the fill value (`5.0::Float64`) and the dimensionality (`(3, 3)::NTuple{2,Int}`). This implies that the compiler can generate very efficient code for any future usage of `A` in the same function.

But now let's say you want to write a function that creates a `3×3×...` array in arbitrary dimensions; you might be tempted to write a function

```
julia> function array3(fillval, N)
           fill(fillval, ntuple(d->3, N))
       end
array3 (generic function with 1 method)

julia> array3(5.0, 2)
3×3 Matrix{Float64}:
 5.0  5.0  5.0
 5.0  5.0  5.0
 5.0  5.0  5.0
```

This works, but (as you can verify for yourself using `@code_warntype array3(5.0, 2)`) the problem is that the output type cannot be inferred: the argument `N` is a *value* of type `Int`, and type-inference does not (and cannot) predict its value in advance. This means that code using the output of this function has to be conservative, checking the type on each access of `A`; such code will be very slow.

Now, one very good way to solve such problems is by using the function-barrier technique. However, in some cases you might want to eliminate the type-instability altogether. In such cases, one approach is to pass the dimensionality as a parameter, for example through `Val{T}()` (see "Value types"):

```
julia> function array3(fillval, ::Val{N}) where N
           fill(fillval, ntuple(d->3, Val(N)))
       end
array3 (generic function with 1 method)

julia> array3(5.0, Val(2))
3×3 Matrix{Float64}:
 5.0  5.0  5.0
 5.0  5.0  5.0
 5.0  5.0  5.0
```

Julia has a specialized version of `ntuple` that accepts a `Val{::Int}` instance as the second parameter; by passing `N` as a type-parameter, you make its "value" known to the compiler. Consequently, this version of `array3` allows the compiler to predict the return type.

However, making use of such techniques can be surprisingly subtle. For example, it would be of no help if you called `array3` from a function like this:

```
function call_array3(fillval, n)
    A = array3(fillval, Val(n))
end
```

Here, you've created the same problem all over again: the compiler can't guess what `n` is, so it doesn't know the *type* of `Val(n)`. Attempting to use `Val`, but doing so incorrectly, can easily make performance *worse* in many situations. (Only in situations where you're effectively combining `Val` with the function-barrier trick, to make the kernel function more efficient, should code like the above be used.)

An example of correct usage of `Val` would be:

```
function filter3(A::AbstractArray{T,N}) where {T,N}
    kernel = array3(1, Val(N))
    filter(A, kernel)
end
```

In this example, `N` is passed as a parameter, so its "value" is known to the compiler. Essentially, `Val(T)` works only when `T` is either hard-coded/literal (`Val(3)`) or already specified in the type-domain.

The dangers of abusing multiple dispatch (aka, more on types with values-as-parameters)

Once one learns to appreciate multiple dispatch, there's an understandable tendency to go overboard and try to use it for everything. For example, you might imagine using it to store information, e.g.

```
struct Car{Make, Model}
    year::Int
    ...more fields...
end
```

and then dispatch on objects like `Car{:Honda,:Accord}(year, args...)`.

This might be worthwhile when either of the following are true:

- You require CPU-intensive processing on each `Car`, and it becomes vastly more efficient if you know the `Make` and `Model` at compile time and the total number of different `Make` or `Model` that will be used is not too large.
- You have homogeneous lists of the same type of `Car` to process, so that you can store them all in an `Array{Car{:Honda,:Accord},N}`.

When the latter holds, a function processing such a homogeneous array can be productively specialized: Julia knows the type of each element in advance (all objects in the container have the same concrete type), so Julia can "look up" the correct method calls when the function is being compiled (obviating the need to check at run-time) and thereby emit efficient code for processing the whole list.

When these do not hold, then it's likely that you'll get no benefit; worse, the resulting "combinatorial explosion of types" will be counterproductive. If `items[i+1]` has a different type than `item[i]`, Julia has to look up the type at run-time, search for the appropriate method in method tables, decide (via type intersection) which one matches, determine whether it has been JIT-compiled yet (and do so if not), and then make the call. In essence, you're asking the full type- system and JIT-compilation machinery to basically execute the equivalent of a switch statement or dictionary lookup in your own code.

Some run-time benchmarks comparing (1) type dispatch, (2) dictionary lookup, and (3) a "switch" statement can be found on discourse.

Perhaps even worse than the run-time impact is the compile-time impact: Julia will compile specialized functions for each different `Car{Make, Model}`; if you have hundreds or thousands of such types, then every function that accepts such an object as a parameter (from a custom `get_year` function you might write yourself, to the generic `push!` function in Julia Base) will have hundreds or thousands of variants compiled for it. Each of these increases the size of the cache of compiled code, the length of internal lists of methods, etc. Excess enthusiasm for values-as-parameters can easily waste enormous resources.

36.4 Memory management and arrays

Pre-allocate outputs

If your function returns an `Array` or some other complex type, it may have to allocate memory. Unfortunately, oftentimes allocation and its converse, garbage collection, are substantial bottlenecks.

Sometimes you can circumvent the need to allocate memory on each function call by preallocating the output. As a trivial example, compare

```
julia> function xinc(x)
           return [x + i for i  in 1:3000]
       end;

julia> function loopinc()
           y = 0
           for i = 1:10^5
               ret = xinc(i)
               y += ret[2]
```

```
            end
            return y
        end;
```

with

```
julia> function xinc!(ret::AbstractVector{T}, x::T) where T
            for i in 1:3000
                ret[i] = x+i
            end
            nothing
        end;

julia> function loopinc_prealloc()
            ret = Vector{Int}(undef, 3000)
            y = 0
            for i = 1:10^5
                xinc!(ret, i)
                y += ret[2]
            end
            return y
        end;
```

Timing results:

```
julia> @time loopinc()
  0.297454 seconds (200.00 k allocations: 2.239 GiB, 39.80% gc time)
5000250000

julia> @time loopinc_prealloc()
  0.009410 seconds (2 allocations: 23.477 KiB)
5000250000
```

Preallocation has other advantages, for example by allowing the caller to control the "output" type from an algorithm. In the example above, we could have passed a `SubArray` rather than an `Array`, had we so desired.

Taken to its extreme, pre-allocation can make your code uglier, so performance measurements and some judgment may be required. However, for "vectorized" (element-wise) functions, the convenient syntax `x .= f.(y)` can be used for in-place operations with fused loops and no temporary arrays (see the dot syntax for vectorizing functions).

Consider using views for slices

In Julia, an array "slice" expression like `array[1:5, :]` creates a copy of that data (except on the left-hand side of an assignment, where `array[1:5, :] = ...` assigns in-place to that portion of `array`). If you are doing many operations on the slice, this can be good for performance because it is more efficient to work with a smaller contiguous copy than it would be to index into the original array. On the other hand, if you are just doing a few simple operations on the slice, the cost of the allocation and copy operations can be substantial.

An alternative is to create a "view" of the array, which is an array object (a `SubArray`) that actually references the data of the original array in-place, without making a copy. (If you write to a view, it modifies the original array's data as well.) This can be done for individual slices by calling `view`, or more simply for a whole expression or block of code by putting `@views` in front of that expression. For example:

```
julia> fcopy(x) = sum(x[2:end-1]);

julia> @views fview(x) = sum(x[2:end-1]);

julia> x = rand(10^6);

julia> @time fcopy(x);
  0.003051 seconds (3 allocations: 7.629 MB)

julia> @time fview(x);
  0.001020 seconds (1 allocation: 16 bytes)
```

Notice both the 3× speedup and the decreased memory allocation of the `fview` version of the function.

Consider StaticArrays.jl for small fixed-size vector/matrix operations

If your application involves many small (`< 100` element) arrays of fixed sizes (i.e. the size is known prior to execution), then you might want to consider using the StaticArrays.jl package. This package allows you to represent such arrays in a way that avoids unnecessary heap allocations and allows the compiler to specialize code for the *size* of the array, e.g. by completely unrolling vector operations (eliminating the loops) and storing elements in CPU registers.

For example, if you are doing computations with 2d geometries, you might have many computations with 2-component vectors. By using the `SVector` type from StaticArrays.jl, you can use convenient vector notation and operations like `norm(3v - w)` on vectors `v` and `w`, while allowing the compiler to unroll the code to a minimal computation equivalent to `@inbounds hypot(3v[1]-w[1], 3v[2]-w[2])`.

More dots: Fuse vectorized operations

Julia has a special dot syntax that converts any scalar function into a "vectorized" function call, and any operator into a "vectorized" operator, with the special property that nested "dot calls" are *fusing*: they are combined at the syntax level into a single loop, without allocating temporary arrays. If you use `.=` and similar assignment operators, the result can also be stored in-place in a pre-allocated array (see above).

In a linear-algebra context, this means that even though operations like `vector + vector` and `vector * scalar` are defined, it can be advantageous to instead use `vector .+ vector` and `vector .* scalar` because the resulting loops can be fused with surrounding computations. For example, consider the two functions:

```
julia> f(x) = 3x.^2 + 4x + 7x.^3;

julia> fdot(x) = @. 3x^2 + 4x + 7x^3; # equivalent to 3 .* x.^2 .+ 4 .* x .+ 7 .* x.^3
```

Both `f` and `fdot` compute the same thing. However, `fdot` (defined with the help of the `@.` macro) is significantly faster when applied to an array:

```
julia> x = rand(10^6);

julia> @time f(x);
  0.019049 seconds (16 allocations: 45.777 MiB, 18.59% gc time)

julia> @time fdot(x);
  0.002790 seconds (6 allocations: 7.630 MiB)

julia> @time f.(x);
  0.002626 seconds (8 allocations: 7.630 MiB)
```

That is, `fdot(x)` is ten times faster and allocates 1/6 the memory of `f(x)`, because each * and + operation in `f(x)` allocates a new temporary array and executes in a separate loop. In this example `f.(x)` is as fast as `fdot(x)` but in many contexts it is more convenient to sprinkle some dots in your expressions than to define a separate function for each vectorized operation.

Fewer dots: Unfuse certain intermediate broadcasts

The dot loop fusion mentioned above enables concise and idiomatic code to express highly performant operations. However, it is important to remember that the fused operation will be computed at every iteration of the broadcast. This means that in some situations, particularly in the presence of composed or multidimensional broadcasts, an expression with dot calls may be computing a function more times than intended. As an example, say we want to build a random matrix whose rows have Euclidean norm one. We might write something like the following:

```
julia> x = rand(1000, 1000);

julia> d = sum(abs2, x; dims=2);

julia> @time x ./= sqrt.(d);
  0.002049 seconds (4 allocations: 96 bytes)
```

This will work. However, this expression will actually recompute `sqrt(d[i])` for *every* element in the row `x[i, :]`, meaning that many more square roots are computed than necessary. To see precisely over which indices the broadcast will iterate, we can call `Broadcast.combine_axes` on the arguments of the fused expression. This will return a tuple of ranges whose entries correspond to the axes of iteration; the product of lengths of these ranges will be the total number of calls to the fused operation.

It follows that when some components of the broadcast expression are constant along an axis—like the `sqrt` along the second dimension in the preceding example—there is potential for a performance improvement by forcibly "unfusing" those components, i.e. allocating the result of the broadcasted operation in advance and reusing the cached value along its constant axis. Some such potential approaches are to use temporary variables, wrap components of a dot expression in `identity`, or use an equivalent intrinsically vectorized (but non-fused) function.

```
julia> @time let s = sqrt.(d); x ./= s end;
  0.000809 seconds (5 allocations: 8.031 KiB)

julia> @time x ./= identity(sqrt.(d));
  0.000608 seconds (5 allocations: 8.031 KiB)
```

```
julia> @time x ./= map(sqrt, d);
  0.000611 seconds (4 allocations: 8.016 KiB)
```

Any of these options yields approximately a three-fold speedup at the cost of an allocation; for large broadcastables this speedup can be asymptotically very large.

Access arrays in memory order, along columns

Multidimensional arrays in Julia are stored in column-major order. This means that arrays are stacked one column at a time. This can be verified using the vec function or the syntax `[:]` as shown below (notice that the array is ordered `[1 3 2 4]`, not `[1 2 3 4]`):

```
julia> x = [1 2; 3 4]
2×2 Matrix{Int64}:
 1  2
 3  4

julia> x[:]
4-element Vector{Int64}:
 1
 3
 2
 4
```

This convention for ordering arrays is common in many languages like Fortran, Matlab, and R (to name a few). The alternative to column-major ordering is row-major ordering, which is the convention adopted by C and Python (numpy) among other languages. Remembering the ordering of arrays can have significant performance effects when looping over arrays. A rule of thumb to keep in mind is that with column-major arrays, the first index changes most rapidly. Essentially this means that looping will be faster if the innermost loop index is the first to appear in a slice expression. Keep in mind that indexing an array with `:` is an implicit loop that iteratively accesses all elements within a particular dimension; it can be faster to extract columns than rows, for example.

Consider the following contrived example. Imagine we wanted to write a function that accepts a `Vector` and returns a square `Matrix` with either the rows or the columns filled with copies of the input vector. Assume that it is not important whether rows or columns are filled with these copies (perhaps the rest of the code can be easily adapted accordingly). We could conceivably do this in at least four ways (in addition to the recommended call to the built-in `repeat`):

```
function copy_cols(x::Vector{T}) where T
    inds = axes(x, 1)
    out = similar(Array{T}, inds, inds)
    for i = inds
        out[:, i] = x
    end
    return out
end

function copy_rows(x::Vector{T}) where T
    inds = axes(x, 1)
```

```
    out = similar(Array{T}, inds, inds)
    for i = inds
        out[i, :] = x
    end
    return out
end

function copy_col_row(x::Vector{T}) where T
    inds = axes(x, 1)
    out = similar(Array{T}, inds, inds)
    for col = inds, row = inds
        out[row, col] = x[row]
    end
    return out
end

function copy_row_col(x::Vector{T}) where T
    inds = axes(x, 1)
    out = similar(Array{T}, inds, inds)
    for row = inds, col = inds
        out[row, col] = x[col]
    end
    return out
end
```

Now we will time each of these functions using the same random `10000` by `1` input vector:

```
julia> x = randn(10000);

julia> fmt(f) = println(rpad(string(f)*": ", 14, ' '), @elapsed f(x))

julia> map(fmt, [copy_cols, copy_rows, copy_col_row, copy_row_col]);
copy_cols:    0.331706323
copy_rows:    1.799009911
copy_col_row: 0.415630047
copy_row_col: 1.721531501
```

Notice that `copy_cols` is much faster than `copy_rows`. This is expected because `copy_cols` respects the column-based memory layout of the `Matrix` and fills it one column at a time. Additionally, `copy_col_row` is much faster than `copy_row_col` because it follows our rule of thumb that the first element to appear in a slice expression should be coupled with the inner-most loop.

Copying data is not always bad

Arrays are stored contiguously in memory, lending themselves to CPU vectorization and fewer memory accesses due to caching. These are the same reasons that it is recommended to access arrays in column-major order (see above). Irregular access patterns and non-contiguous views can drastically slow down computations on arrays because of non-sequential memory access.

Copying irregularly-accessed data into a contiguous array before repeatedly accessing it can result in a large speedup, such as in the example below. Here, a matrix is being accessed at randomly-shuffled indices

before being multiplied. Copying into plain arrays speeds up the multiplication even with the added cost of copying and allocation.

```
julia> using Random

julia> A = randn(3000, 3000);

julia> x = randn(2000);

julia> inds = shuffle(1:3000)[1:2000];

julia> function iterated_neural_network(A, x, depth)
           for _ in 1:depth
               x .= max.(0, A * x)
           end
           argmax(x)
       end

julia> @time iterated_neural_network(view(A, inds, inds), x, 10)
  0.324903 seconds (12 allocations: 157.562 KiB)
1569

julia> @time iterated_neural_network(A[inds, inds], x, 10)
  0.054576 seconds (13 allocations: 30.671 MiB, 13.33% gc time)
1569
```

Provided there is enough memory, the cost of copying the view to an array is outweighed by the speed boost from doing the repeated matrix multiplications on a contiguous array.

Multithreading and linear algebra

This section applies to multithreaded Julia code which, in each thread, performs linear algebra operations. Indeed, these linear algebra operations involve BLAS / LAPACK calls, which are themselves multithreaded. In this case, one must ensure that cores aren't oversubscribed due to the two different types of multithreading.

Julia compiles and uses its own copy of OpenBLAS for linear algebra, whose number of threads is controlled by the environment variable `OPENBLAS_NUM_THREADS`. It can either be set as a command line option when launching Julia, or modified during the Julia session with `BLAS.set_num_threads(N)` (the submodule `BLAS` is exported by `using LinearAlgebra`). Its current value can be accessed with `BLAS.get_num_threads()`.

When the user does not specify anything, Julia tries to choose a reasonable value for the number of OpenBLAS threads (e.g. based on the platform, the Julia version, etc.). However, it is generally recommended to check and set the value manually. The OpenBLAS behavior is as follows:

- If `OPENBLAS_NUM_THREADS=1`, OpenBLAS uses the calling Julia thread(s), i.e. it "lives in" the Julia thread that runs the computation.
- If `OPENBLAS_NUM_THREADS=N>1`, OpenBLAS creates and manages its own pool of threads (`N` in total). There is just one OpenBLAS thread pool shared among all Julia threads.

When you start Julia in multithreaded mode with `JULIA_NUM_THREADS=X`, it is generally recommended to set `OPENBLAS_NUM_THREADS=1`. Given the behavior described above, increasing the number of BLAS threads to `N>1` can very easily lead to worse performance, in particular when `N<<X`. However this is just a rule of thumb, and the best way to set each number of threads is to experiment on your specific application.

Alternative linear algebra backends

As an alternative to OpenBLAS, there exist several other backends that can help with linear algebra performance. Prominent examples include MKL.jl and AppleAccelerate.jl.

These are external packages, so we will not discuss them in detail here. Please refer to their respective documentations (especially because they have different behaviors than OpenBLAS with respect to multithreading).

36.5 Execution latency, package loading and package precompiling time

Reducing time to first plot etc.

The first time a julia method is called it (and any methods it calls, or ones that can be statically determined) will be compiled. The `@time` macro family illustrates this.

```
julia> foo() = rand(2,2) * rand(2,2)
foo (generic function with 1 method)

julia> @time @eval foo();
  0.252395 seconds (1.12 M allocations: 56.178 MiB, 2.93% gc time, 98.12% compilation time)

julia> @time @eval foo();
  0.000156 seconds (63 allocations: 2.453 KiB)
```

Note that `@time @eval` is better for measuring compilation time because without `@eval`, some compilation may already be done before timing starts.

When developing a package, you may be able to improve the experience of your users with *precompilation* so that when they use the package, the code they use is already compiled. To precompile package code effectively, it's recommended to use `PrecompileTools.jl` to run a "precompile workload" during precompilation time that is representative of typical package usage, which will cache the native compiled code into the package `pkgimage` cache, greatly reducing "time to first execution" (often referred to as TTFX) for such usage.

Note that `PrecompileTools.jl` workloads can be disabled and sometimes configured via Preferences if you do not want to spend the extra time precompiling, which may be the case during development of a package.

Reducing package loading time

Keeping the time taken to load the package down is usually helpful. General good practice for package developers includes:

1. Reduce your dependencies to those you really need. Consider using package extensions to support interoperability with other packages without bloating your essential dependencies.

2. Avoid use of `__init__()` functions unless there is no alternative, especially those which might trigger a lot of compilation, or just take a long time to execute.

3. Where possible, fix invalidations among your dependencies and from your package code.

The tool `@time_imports` can be useful in the REPL to review the above factors.

```
julia> @time @time_imports using Plots
      0.5 ms  Printf
     16.4 ms  Dates
      0.7 ms  Statistics
                ┌ 23.8 ms SuiteSparse_jll.__init__() 86.11% compilation time (100% recompilation)
     90.1 ms  SuiteSparse_jll 91.57% compilation time (82% recompilation)
      0.9 ms  Serialization
                ┌ 39.8 ms SparseArrays.CHOLMOD.__init__() 99.47% compilation time (100%
                ↪  recompilation)
    166.9 ms  SparseArrays 23.74% compilation time (100% recompilation)
      0.4 ms  Statistics → SparseArraysExt
      0.5 ms  TOML
      8.0 ms  Preferences
      0.3 ms  PrecompileTools
      0.2 ms  Reexport
... many deps omitted for example ...
      1.4 ms  Tar
                ┌ 73.8 ms p7zip_jll.__init__() 99.93% compilation time (100% recompilation)
     79.4 ms  p7zip_jll 92.91% compilation time (100% recompilation)
                ┌ 27.7 ms GR.GRPreferences.__init__() 99.77% compilation time (100% recompilation)
     43.0 ms  GR 64.26% compilation time (100% recompilation)
                ┌ 2.1 ms Plots.__init__() 91.80% compilation time (100% recompilation)
    300.9 ms  Plots 0.65% compilation time (100% recompilation)
  1.795602 seconds (3.33 M allocations: 190.153 MiB, 7.91% gc time, 39.45% compilation time: 97%
  ↪  of which was recompilation)
```

Notice that in this example there are multiple packages loaded, some with `__init__()` functions, some of which cause compilation of which some is recompilation. Recompilation is caused by earlier packages invalidating methods, then in these cases when the following packages run their `__init__()` function some hit recompilation before the code can be run.

Further, note the `Statistics` extension `SparseArraysExt` has been activated because `SparseArrays` is in the dependency tree. i.e. see `0.4 ms Statistics → SparseArraysExt`.

This report gives a good opportunity to review whether the cost of dependency load time is worth the functionality it brings. Also the Pkg utility `why` can be used to report why an indirect dependency exists.

```
(CustomPackage) pkg> why FFMPEG_jll
  Plots → FFMPEG → FFMPEG_jll
  Plots → GR → GR_jll → FFMPEG_jll
```

or to see the indirect dependencies that a package brings in, you can `pkg> rm` the package, see the deps that are removed from the manifest, then revert the change with `pkg> undo`.

If loading time is dominated by slow `__init__()` methods having compilation, one verbose way to identify what is being compiled is to use the julia args `--trace-compile=stderr --trace-compile-timing` which will report a `precompile` statement each time a method is compiled, along with how long compilation took. The InteractiveUtils macro `@trace_compile` provides a way to enable those args for a specific call. So a call for a complete report report would look like:

```
julia> @time @time_imports @trace_compile using CustomPackage
...
```

Note the `--startup-file=no` which helps isolate the test from packages you may have in your `startup.jl`.

More analysis of the reasons for recompilation can be achieved with the `SnoopCompile` package.

Tracing expression evaluation

If you need to understand what code is being evaluated during test or script execution, you can use the `--trace-eval` command-line option or the `Base.TRACE_EVAL` global control to trace the outermost expressions being evaluated (top-level statements). Note this does not individually report the contents of function calls or code blocks:

```
# Show only location information during evaluation
julia --trace-eval=loc script.jl

# Show full expressions being evaluated
julia --trace-eval=full script.jl
```

You can also control this programmatically:

```
# Enable full expression tracing
Base.TRACE_EVAL = :full

# Show only locations
Base.TRACE_EVAL = :loc

# Disable tracing
Base.TRACE_EVAL = :no

# Reset to use command-line setting
Base.TRACE_EVAL = nothing
```

Reducing precompilation time

If package precompilation is taking a long time, one option is to set the following internal and then precompile.

```
julia> Base.PRECOMPILE_TRACE_COMPILE[] = "stderr"

pkg> precompile
```

This has the effect of setting `--trace-compile=stderr --trace-compile-timing` in the precompilation processes themselves, so will show which methods are precompiled and how long they took to precompile.

There are also profiling options such as using the external profiler Tracy to profile the precompilation process.

36.6 Miscellaneous

Tweaks

These are some minor points that might help in tight inner loops.

- Avoid unnecessary arrays. For example, instead of `sum([x,y,z])` use x+y+z.
- Use `abs2(z)` instead of `abs(z)^2` for complex z. In general, try to rewrite code to use `abs2` instead of `abs` for complex arguments.
- Use `div(x,y)` for truncating division of integers instead of `trunc(x/y)`, `fld(x,y)` instead of `floor(x/y)`, and `cld(x,y)` instead of `ceil(x/y)`.

Fix deprecation warnings

A deprecated function internally performs a lookup in order to print a relevant warning only once. This extra lookup can cause a significant slowdown, so all uses of deprecated functions should be modified as suggested by the warnings.

Performance Annotations

Sometimes you can enable better optimization by promising certain program properties.

- Use `@inbounds` to eliminate array bounds checking within expressions. Be certain before doing this. If the indices are ever out of bounds, you may suffer crashes or silent corruption.
- Use `@fastmath` to allow floating point optimizations that are correct for real numbers, but lead to differences for IEEE numbers. Be careful when doing this, as this may change numerical results. This corresponds to the `-ffast-math` option of clang.
- Write `@simd` in front of `for` loops to promise that the iterations are independent and may be reordered. Note that in many cases, Julia can automatically vectorize code without the `@simd` macro; it is only beneficial in cases where such a transformation would otherwise be illegal, including cases like allowing floating-point re-associativity and ignoring dependent memory accesses (`@simd ivdep`). Again, be very careful when asserting `@simd` as erroneously annotating a loop with dependent iterations may result in unexpected results. In particular, note that `setindex!` on some `AbstractArray` subtypes is inherently dependent upon iteration order. **This feature is experimental** and could change or disappear in future versions of Julia.

The common idiom of using 1:n to index into an AbstractArray is not safe if the Array uses unconventional indexing, and may cause a segmentation fault if bounds checking is turned off. Use `LinearIndices(x)` or `eachindex(x)` instead (see also Arrays with custom indices).

Note

While `@simd` needs to be placed directly in front of an innermost `for` loop, both `@inbounds` and `@fastmath` can be applied to either single expressions or all the expressions that appear within nested blocks of code, e.g., using `@inbounds begin` or `@inbounds for ...`.

Here is an example with both `@inbounds` and `@simd` markup (we here use `@noinline` to prevent the optimizer from trying to be too clever and defeat our benchmark):

```
@noinline function inner(x, y)
    s = zero(eltype(x))
    for i in eachindex(x, y)
        @inbounds s += x[i]*y[i]
    end
    return s
end

@noinline function innersimd(x, y)
    s = zero(eltype(x))
    @simd for i in eachindex(x, y)
        @inbounds s += x[i] * y[i]
    end
    return s
end

function timeit(n, reps)
    x = rand(Float32, n)
    y = rand(Float32, n)
    s = zero(Float64)
    time = @elapsed for j in 1:reps
        s += inner(x, y)
    end
    println("GFlop/sec        = ", 2n*reps / time*1E-9)
    time = @elapsed for j in 1:reps
        s += innersimd(x, y)
    end
    println("GFlop/sec (SIMD) = ", 2n*reps / time*1E-9)
end

timeit(1000, 1000)
```

On a computer with a 2.4GHz Intel Core i5 processor, this produces:

```
GFlop/sec        = 1.9467069505224963
GFlop/sec (SIMD) = 17.578554163920018
```

(GFlop/sec measures the performance, and larger numbers are better.)

Here is an example with all three kinds of markup. This program first calculates the finite difference of a one-dimensional array, and then evaluates the L2-norm of the result:

```
function init!(u::Vector)
    n = length(u)
    dx = 1.0 / (n-1)
    @fastmath @inbounds @simd for i in 1:n #by asserting that `u` is a `Vector` we can assume it
    ↪  has 1-based indexing
        u[i] = sin(2pi*dx*i)
    end
end

function deriv!(u::Vector, du)
    n = length(u)
    dx = 1.0 / (n-1)
    @fastmath @inbounds du[1] = (u[2] - u[1]) / dx
    @fastmath @inbounds @simd for i in 2:n-1
        du[i] = (u[i+1] - u[i-1]) / (2*dx)
    end
    @fastmath @inbounds du[n] = (u[n] - u[n-1]) / dx
end

function mynorm(u::Vector)
    n = length(u)
    T = eltype(u)
    s = zero(T)
    @fastmath @inbounds @simd for i in 1:n
        s += u[i]^2
    end
    @fastmath @inbounds return sqrt(s)
end

function main()
    n = 2000
    u = Vector{Float64}(undef, n)
    init!(u)
    du = similar(u)

    deriv!(u, du)
    nu = mynorm(du)

    @time for i in 1:10^6
        deriv!(u, du)
        nu = mynorm(du)
    end

    println(nu)
end

main()
```

On a computer with a 2.7 GHz Intel Core i7 processor, this produces:

```
$ julia wave.jl;
  1.207814709 seconds
4.443986180758249
```

```
$ julia --math-mode=ieee wave.jl;
  4.487083643 seconds
4.443986180758249
```

Here, the option `--math-mode=ieee` disables the `@fastmath` macro, so that we can compare results.

In this case, the speedup due to `@fastmath` is a factor of about 3.7. This is unusually large – in general, the speedup will be smaller. (In this particular example, the working set of the benchmark is small enough to fit into the L1 cache of the processor, so that memory access latency does not play a role, and computing time is dominated by CPU usage. In many real world programs this is not the case.) Also, in this case this optimization does not change the result – in general, the result will be slightly different. In some cases, especially for numerically unstable algorithms, the result can be very different.

The annotation `@fastmath` re-arranges floating point expressions, e.g. changing the order of evaluation, or assuming that certain special cases (inf, nan) cannot occur. In this case (and on this particular computer), the main difference is that the expression `1 / (2*dx)` in the function `deriv` is hoisted out of the loop (i.e. calculated outside the loop), as if one had written `idx = 1 / (2*dx)`. In the loop, the expression `... / (2*dx)` then becomes `... * idx`, which is much faster to evaluate. Of course, both the actual optimization that is applied by the compiler as well as the resulting speedup depend very much on the hardware. You can examine the change in generated code by using Julia's `code_native` function.

Note that `@fastmath` also assumes that `NaN`s will not occur during the computation, which can lead to surprising behavior:

```
julia> f(x) = isnan(x);

julia> f(NaN)
true

julia> f_fast(x) = @fastmath isnan(x);

julia> f_fast(NaN)
false
```

Treat Subnormal Numbers as Zeros

Subnormal numbers, formerly called denormal numbers, are useful in many contexts, but incur a performance penalty on some hardware. A call `set_zero_subnormals(true)` grants permission for floating-point operations to treat subnormal inputs or outputs as zeros, which may improve performance on some hardware. A call `set_zero_subnormals(false)` enforces strict IEEE behavior for subnormal numbers.

Below is an example where subnormals noticeably impact performance on some hardware:

```
function timestep(b::Vector{T}, a::Vector{T}, Δt::T) where T
    @assert length(a)==length(b)
    n = length(b)
    b[1] = 1                            # Boundary condition
    for i=2:n-1
        b[i] = a[i] + (a[i-1] - T(2)*a[i] + a[i+1]) * Δt
    end
    b[n] = 0                            # Boundary condition
```

```
end

function heatflow(a::Vector{T}, nstep::Integer) where T
    b = similar(a)
    for t=1:div(nstep,2)                # Assume nstep is even
        timestep(b,a,T(0.1))
        timestep(a,b,T(0.1))
    end
end

heatflow(zeros(Float32,10),2)           # Force compilation
for trial=1:6
    a = zeros(Float32,1000)
    set_zero_subnormals(iseven(trial))  # Odd trials use strict IEEE arithmetic
    @time heatflow(a,1000)
end
```

This gives an output similar to

```
  0.002202 seconds (1 allocation: 4.063 KiB)
  0.001502 seconds (1 allocation: 4.063 KiB)
  0.002139 seconds (1 allocation: 4.063 KiB)
  0.001454 seconds (1 allocation: 4.063 KiB)
  0.002115 seconds (1 allocation: 4.063 KiB)
  0.001455 seconds (1 allocation: 4.063 KiB)
```

Note how each even iteration is significantly faster.

This example generates many subnormal numbers because the values in a become an exponentially decreasing curve, which slowly flattens out over time.

Treating subnormals as zeros should be used with caution, because doing so breaks some identities, such as `x-y == 0` implies `x == y`:

```
julia> x = 3f-38; y = 2f-38;

julia> set_zero_subnormals(true); (x - y, x == y)
(0.0f0, false)

julia> set_zero_subnormals(false); (x - y, x == y)
(1.0000001f-38, false)
```

In some applications, an alternative to zeroing subnormal numbers is to inject a tiny bit of noise. For example, instead of initializing a with zeros, initialize it with:

```
a = rand(Float32,1000) * 1.f-9
```

Avoid string interpolation for I/O

When writing data to a file (or other I/O device), forming extra intermediate strings is a source of overhead. Instead of:

```
println(file, "$a $b")
```

use:

```
println(file, a, " ", b)
```

The first version of the code forms a string, then writes it to the file, while the second version writes values directly to the file. Also notice that in some cases string interpolation can be harder to read. Consider:

```
println(file, "$(f(a))$(f(b))")
```

versus:

```
println(file, f(a), f(b))
```

Avoid eager string materialization

In settings where a string representation of an object is only needed conditionally (e.g. in error paths of functions or conditional warnings such as deprecations), it is advisable to avoid the overhead of eagerly materializing the string. Since Julia 1.8, this can be achieved via `LazyString` and the corresponding string macro `@lazy_str`.

For example, instead of:

```
Base.depwarn("`foo` is deprecated for type $(typeof(x))", :bar)
```

use:

```
Base.depwarn(lazy"`foo` is deprecated for type $(typeof(x))", :bar)
```

or the equivalent macro-free version:

```
Base.depwarn(LazyString("`foo` is deprecated for type ", typeof(x)), :bar)
```

Through this approach, the interpolated string will only be constructed when it is actually displayed.

Optimize network I/O during parallel execution

When executing a remote function in parallel:

```
using Distributed

responses = Vector{Any}(undef, nworkers())
@sync begin
    for (idx, pid) in enumerate(workers())
        Threads.@spawn responses[idx] = remotecall_fetch(foo, pid, args...)
    end
end
```

is faster than:

```
using Distributed

refs = Vector{Any}(undef, nworkers())
for (idx, pid) in enumerate(workers())
    refs[idx] = @spawnat pid foo(args...)
end
responses = [fetch(r) for r in refs]
```

The former results in a single network round-trip to every worker, while the latter results in two network calls - first by the `@spawnat` and the second due to the `fetch` (or even a `wait`). The `fetch`/`wait` is also being executed serially resulting in an overall poorer performance.

Use `MutableArithmetics` for more control over allocation for mutable arithmetic types

Some `Number` subtypes, such as `BigInt` or `BigFloat`, may be implemented as `mutable struct` types, or they may have mutable components. The arithmetic interfaces in Julia Base usually opt for convenience over efficiency in such cases, so using them in a naive manner may result in suboptimal performance. The abstractions of the `MutableArithmetics` package, on the other hand, make it possible to exploit the mutability of such types for writing fast code that allocates only as much as necessary. `MutableArithmetics` also makes it possible to copy values of mutable arithmetic types explicitly when necessary. `MutableArithmetics` is a user package and is not affiliated with the Julia project.

Chapter 37

Workflow Tips

Here are some tips for working with Julia efficiently.

37.1 REPL-based workflow

As already elaborated in The Julia REPL, Julia's REPL provides rich functionality that facilitates an efficient interactive workflow. Here are some tips that might further enhance your experience at the command line.

A basic editor/REPL workflow

The most basic Julia workflows involve using a text editor in conjunction with the `julia` command line.

Create a file, say `Tmp.jl`, and include within it

```
module Tmp

say_hello() = println("Hello!")

# Your other definitions here

end # module

using .Tmp
```

Then, in the same directory, start the Julia REPL (using the `julia` command). Run the new file as follows:

```
julia> include("Tmp.jl")

julia> Tmp.say_hello()
Hello!
```

Explore ideas in the REPL. Save good ideas in `Tmp.jl`. To reload the file after it has been changed, just `include` it again.

The key in the above is that your code is encapsulated in a module. That allows you to edit `struct` definitions and remove methods, without restarting Julia.

(Explanation: `structs` cannot be edited after definition, nor can methods be deleted. But you *can* overwrite the definition of a module, which is what we do when we re-`include("Tmp.jl")`).

In addition, the encapsulation of code in a module protects it from being influenced by previous state in the REPL, protecting you from hard-to-detect errors.

37.2 Browser-based workflow

There are a few ways to interact with Julia in a browser:

- Using Pluto notebooks through Pluto.jl
- Using Jupyter notebooks through IJulia.jl

37.3 Revise-based workflows

Whether you're at the REPL or in IJulia, you can typically improve your development experience with Revise. It is common to configure Revise to start whenever julia is started, as per the instructions in the Revise documentation. Once configured, Revise will track changes to files in any loaded modules, and to any files loaded in to the REPL with `includet` (but not with plain `include`); you can then edit the files and the changes take effect without restarting your julia session. A standard workflow is similar to the REPL-based workflow above, with the following modifications:

1. Put your code in a module somewhere on your load path. There are several options for achieving this, of which two recommended choices are:

   - For long-term projects, use PkgTemplates:

     ```
     using PkgTemplates
     t = Template()
     t("MyPkg")
     ```

     This will create a blank package, `"MyPkg"`, in your `.julia/dev` directory. Note that PkgTemplates allows you to control many different options through its `Template` constructor.

     In step 2 below, edit `MyPkg/src/MyPkg.jl` to change the source code, and `MyPkg/test/runtests.jl` for the tests.

   - For "throw-away" projects, you can avoid any need for cleanup by doing your work in your temporary directory (e.g., `/tmp`).

     Navigate to your temporary directory and launch Julia, then do the following:

     ```
     pkg> generate MyPkg            # type ] to enter pkg mode
     julia> push!(LOAD_PATH, pwd())   # hit backspace to exit pkg mode
     ```

     If you restart your Julia session you'll have to re-issue that command modifying `LOAD_PATH`.

     In step 2 below, edit `MyPkg/src/MyPkg.jl` to change the source code, and create any test file of your choosing.

2. Develop your package

   Before loading any code, make sure you're running Revise: say `using Revise` or follow its documentation on configuring it to run automatically.

   Then navigate to the directory containing your test file (here assumed to be `"runtests.jl"`) and do the following:

```
julia> using MyPkg

julia> include("runtests.jl")
```

You can iteratively modify the code in MyPkg in your editor and re-run the tests with `include("runtests.jl")`. You generally should not need to restart your Julia session to see the changes take effect (subject to a few limitations).

Chapter 38

Style Guide

The following sections explain a few aspects of idiomatic Julia coding style. None of these rules are absolute; they are only suggestions to help familiarize you with the language and to help you choose among alternative designs.

38.1 Indentation

Use 4 spaces per indentation level.

38.2 Write functions, not just scripts

Writing code as a series of steps at the top level is a quick way to get started solving a problem, but you should try to divide a program into functions as soon as possible. Functions are more reusable and testable, and clarify what steps are being done and what their inputs and outputs are. Furthermore, code inside functions tends to run much faster than top level code, due to how Julia's compiler works.

It is also worth emphasizing that functions should take arguments, instead of operating directly on global variables (aside from constants like `pi`).

38.3 Write docstrings

Comments describing an object should typically be written as docstrings for editor and REPL accessibility. Inline comments (`# comment`) and multiline comments (`#= comment =#`) are appropriate for information that is intended only for the reader of the code (as opposed to a user).

38.4 Avoid writing overly-specific types

Code should be as generic as possible. Instead of writing:

```
Complex{Float64}(x)
```

it's better to use available generic functions:

```
complex(float(x))
```

The second version will convert x to an appropriate type, instead of always the same type.

This style point is especially relevant to function arguments. For example, don't declare an argument to be of type `Int` or `Int32` if it really could be any integer, expressed with the abstract type `Integer`. In fact, in many cases you can omit the argument type altogether, unless it is needed to disambiguate from other method definitions, since a `MethodError` will be thrown anyway if a type is passed that does not support any of the requisite operations. (This is known as duck typing.)

For example, consider the following definitions of a function addone that returns one plus its argument:

```
addone(x::Int) = x + 1                  # works only for Int
addone(x::Integer) = x + oneunit(x)     # any integer type
addone(x::Number) = x + oneunit(x)      # any numeric type
addone(x) = x + oneunit(x)              # any type supporting + and oneunit
```

The last definition of addone handles any type supporting `oneunit` (which returns 1 in the same type as x, which avoids unwanted type promotion) and the `+` function with those arguments. The key thing to realize is that there is *no performance penalty* to defining *only* the general `addone(x) = x + oneunit(x)`, because Julia will automatically compile specialized versions as needed. For example, the first time you call `addone(12)`, Julia will automatically compile a specialized addone function for `x::Int` arguments, with the call to `oneunit` replaced by its inlined value 1. Therefore, the first three definitions of addone above are completely redundant with the fourth definition.

38.5 Handle excess argument diversity in the caller

Instead of:

```
function foo(x, y)
    x = Int(x); y = Int(y)
    ...
end
foo(x, y)
```

use:

```
function foo(x::Int, y::Int)
    ...
end
foo(Int(x), Int(y))
```

This is better style because `foo` does not really accept numbers of all types; it really needs `Int` s.

One issue here is that if a function inherently requires integers, it might be better to force the caller to decide how non-integers should be converted (e.g. floor or ceiling). Another issue is that declaring more specific types leaves more "space" for future method definitions.

38.6 Append ! to names of functions that modify their arguments

Instead of:

```
function double(a::AbstractArray{<:Number})
    for i in eachindex(a)
        a[i] *= 2
    end
    return a
end
```

use:

```
function double!(a::AbstractArray{<:Number})
    for i in eachindex(a)
        a[i] *= 2
    end
    return a
end
```

Julia Base uses this convention throughout and contains examples of functions with both copying and modifying forms (e.g., `sort` and `sort!`), and others which are just modifying (e.g., `push!`, `pop!`, `splice!`). It is typical for such functions to also return the modified array for convenience.

Functions related to IO or making use of random number generators (RNG) are notable exceptions: Since these functions almost invariably must mutate the IO or RNG, functions ending with `!` are used to signify a mutation *other* than mutating the IO or advancing the RNG state. For example, `rand(x)` mutates the RNG, whereas `rand!(x)` mutates both the RNG and `x`; similarly, `read(io)` mutates `io`, whereas `read!(io, x)` mutates both arguments.

38.7 Avoid strange type `Unions`

Types such as `Union{Function,AbstractString}` are often a sign that some design could be cleaner.

38.8 Avoid elaborate container types

It is usually not much help to construct arrays like the following:

```
a = Vector{Union{Int,AbstractString,Tuple,Array}}(undef, n)
```

In this case `Vector{Any}(undef, n)` is better. It is also more helpful to the compiler to annotate specific uses (e.g. `a[i]::Int`) than to try to pack many alternatives into one type.

38.9 Prefer exported methods over direct field access

Idiomatic Julia code should generally treat a module's exported methods as the interface to its types. An object's fields are generally considered implementation details and user code should only access them directly if this is stated to be the API. This has several benefits:

- Package developers are freer to change the implementation without breaking user code.
- Methods can be passed to higher-order constructs like `map` (e.g. `map(imag, zs)`) rather than `[z.im for z in zs]`).
- Methods can be defined on abstract types.
- Methods can describe a conceptual operation that can be shared across disparate types (e.g. `real(z)` works on Complex numbers or Quaternions).

Julia's dispatch system encourages this style because `play(x::MyType)` only defines the `play` method on that particular type, leaving other types to have their own implementation.

Similarly, non-exported functions are typically internal and subject to change, unless the documentations states otherwise. Names sometimes are given a _ prefix (or suffix) to further suggest that something is "internal" or an implementation-detail, but it is not a rule.

Counter-examples to this rule include `NamedTuple`, `RegexMatch`, `StatStruct`.

38.10 Use naming conventions consistent with Julia base/

- modules and type names use capitalization and camel case: `module SparseArrays`, `struct UnitRange`.
- constants use all uppercase and underscores (`LOAD_PATH`, `VERSION`).
- while anything not marked with `public` or `export` is considered internal, a prefix of _ also indicates that an object is not intended for public use.
- functions mutating at least one of their arguments end in `!`.
- conciseness is valued, but avoid abbreviation (`indexin` rather than `indxin`) as it becomes difficult to remember whether and how particular words are abbreviated.

If a function name requires multiple words, consider whether it might represent more than one concept and might be better split into pieces.

Function names should be written in snake case (`minimum`, `count_zeros`, `escape_string`). Base often breaks this convention by squashing words together (`splitpath`, `readeach`) but this style is not recommended for packages.

38.11 Write functions with argument ordering similar to Julia Base

As a general rule, the Base library uses the following order of arguments to functions, as applicable:

1. **Function argument**. Putting a function argument first permits the use of `do` blocks for passing multiline anonymous functions.
2. **I/O stream**. Specifying the `IO` object first permits passing the function to functions such as `sprint`, e.g. `sprint(show, x)`.
3. **Input being mutated**. For example, in `fill!(x, v)`, x is the object being mutated and it appears before the value to be inserted into x.

4. **Type**. Passing a type typically means that the output will have the given type. In `parse(Int, "1")`, the type comes before the string to parse. There are many such examples where the type appears first, but it's useful to note that in `read(io, String)`, the `IO` argument appears before the type, which is in keeping with the order outlined here.
5. **Input not being mutated**. In `fill!(x, v)`, `v` is *not* being mutated and it comes after `x`.
6. **Key**. For associative collections, this is the key of the key-value pair(s). For other indexed collections, this is the index.
7. **Value**. For associative collections, this is the value of the key-value pair(s). In cases like `fill!(x, v)`, this is `v`.
8. **Everything else**. Any other arguments.
9. **Varargs**. This refers to arguments that can be listed indefinitely at the end of a function call. For example, in `Matrix{T}(undef, dims)`, the dimensions can be given as a `Tuple`, e.g. `Matrix{T}(undef, (1,2))`, or as `Vararg`s, e.g. `Matrix{T}(undef, 1, 2)`.
10. **Keyword arguments**. In Julia keyword arguments have to come last anyway in function definitions; they're listed here for the sake of completeness.

The vast majority of functions will not take every kind of argument listed above; the numbers merely denote the precedence that should be used for any applicable arguments to a function.

There are of course a few exceptions. For example, in `convert`, the type should always come first. In `setindex!`, the value comes before the indices so that the indices can be provided as varargs.

When designing APIs, adhering to this general order as much as possible is likely to give users of your functions a more consistent experience.

38.12 Don't overuse try-catch

It is better to avoid errors than to rely on catching them.

38.13 Don't parenthesize conditions

Julia doesn't require parens around conditions in `if` and `while`. Write:

```
if a == b
```

instead of:

```
if (a == b)
```

38.14 Don't overuse ...

Splicing function arguments can be addictive. Instead of `[a..., b...]`, use simply `[a; b]`, which already concatenates arrays. `collect(a)` is better than `[a...]`, but since `a` is already iterable it is often even better to leave it alone, and not convert it to an array.

38.15 Ensure constructors return an instance of their own type

When a method `T(x)` is called on a type `T`, it is generally expected to return a value of type `T`. Defining a constructor that returns an unexpected type can lead to confusing and unpredictable behavior:

```
julia> struct Foo{T}
           x::T
       end

julia> Base.Float64(foo::Foo) = Foo(Float64(foo.x))  # Do not define methods like this

julia> Float64(Foo(3))  # Should return `Float64`
Foo{Float64}(3.0)

julia> Foo{Int}(x) = Foo{Float64}(x)  # Do not define methods like this

julia> Foo{Int}(3)  # Should return `Foo{Int}`
Foo{Float64}(3.0)
```

To maintain code clarity and ensure type consistency, always design constructors to return an instance of the type they are supposed to construct.

38.16 Don't use unnecessary static parameters

A function signature:

```
foo(x::T) where {T<:Real} = ...
```

should be written as:

```
foo(x::Real) = ...
```

instead, especially if `T` is not used in the function body. Even if `T` is used, it can be replaced with `typeof(x)` if convenient. There is no performance difference. Note that this is not a general caution against static parameters, just against uses where they are not needed.

Note also that container types, specifically may need type parameters in function calls. See the FAQ Avoid fields with abstract containers for more information.

38.17 Avoid confusion about whether something is an instance or a type

Sets of definitions like the following are confusing:

```
foo(::Type{MyType}) = ...
foo(::MyType) = foo(MyType)
```

Decide whether the concept in question will be written as `MyType` or `MyType()`, and stick to it.

The preferred style is to use instances by default, and only add methods involving `Type{MyType}` later if they become necessary to solve some problems.

If a type is effectively an enumeration, it should be defined as a single (ideally immutable struct or primitive) type, with the enumeration values being instances of it. Constructors and conversions can check whether values are valid. This design is preferred over making the enumeration an abstract type, with the "values" as subtypes.

38.18 Don't overuse macros

Be aware of when a macro could really be a function instead.

Calling `eval` inside a macro is a particularly dangerous warning sign; it means the macro will only work when called at the top level. If such a macro is written as a function instead, it will naturally have access to the run-time values it needs.

38.19 Don't expose unsafe operations at the interface level

If you have a type that uses a native pointer:

```
mutable struct NativeType
    p::Ptr{UInt8}
    ...
end
```

don't write definitions like the following:

```
getindex(x::NativeType, i) = unsafe_load(x.p, i)
```

The problem is that users of this type can write `x[i]` without realizing that the operation is unsafe, and then be susceptible to memory bugs.

Such a function should either check the operation to ensure it is safe, or have `unsafe` somewhere in its name to alert callers.

38.20 Don't overload methods of base container types

It is possible to write definitions like the following:

```
show(io::IO, v::Vector{MyType}) = ...
```

This would provide custom showing of vectors with a specific new element type. While tempting, this should be avoided. The trouble is that users will expect a well-known type like `Vector()` to behave in a certain way, and overly customizing its behavior can make it harder to work with.

38.21 Avoid type piracy

"Type piracy" refers to the practice of extending or redefining methods in Base or other packages on types that you have not defined. In extreme cases, you can crash Julia (e.g. if your method extension or redefinition causes invalid input to be passed to a `ccall`). Type piracy can complicate reasoning about code, and may introduce incompatibilities that are hard to predict and diagnose.

As an example, suppose you wanted to define multiplication on symbols in a module:

```
module A
import Base.*
*(x::Symbol, y::Symbol) = Symbol(x,y)
end
```

The problem is that now any other module that uses `Base.*` will also see this definition. Since `Symbol` is defined in Base and is used by other modules, this can change the behavior of unrelated code unexpectedly. There are several alternatives here, including using a different function name, or wrapping the `Symbols` in another type that you define.

Sometimes, coupled packages may engage in type piracy to separate features from definitions, especially when the packages were designed by collaborating authors, and when the definitions are reusable. For example, one package might provide some types useful for working with colors; another package could define methods for those types that enable conversions between color spaces. Another example might be a package that acts as a thin wrapper for some C code, which another package might then pirate to implement a higher-level, Julia-friendly API.

38.22 Be careful with type equality

You generally want to use `isa` and `<:` for testing types, not ==. Checking types for exact equality typically only makes sense when comparing to a known concrete type (e.g. `T == Float64`), or if you *really, really* know what you're doing.

38.23 Don't write a trivial anonymous function `x->f(x)` for a named function `f`

Since higher-order functions are often called with anonymous functions, it is easy to conclude that this is desirable or even necessary. But any function can be passed directly, without being "wrapped" in an anonymous function. Instead of writing `map(x->f(x), a)`, write `map(f, a)`.

38.24 Avoid using floats for numeric literals in generic code when possible

If you write generic code which handles numbers, and which can be expected to run with many different numeric type arguments, try using literals of a numeric type that will affect the arguments as little as possible through promotion.

For example,

```
julia> f(x) = 2.0 * x
f (generic function with 1 method)

julia> f(1//2)
```

```
1.0

julia> f(1/2)
1.0

julia> f(1)
2.0
```

while

```
julia> g(x) = 2 * x
g (generic function with 1 method)

julia> g(1//2)
1//1

julia> g(1/2)
1.0

julia> g(1)
2
```

As you can see, the second version, where we used an `Int` literal, preserved the type of the input argument, while the first didn't. This is because e.g. `promote_type(Int, Float64) == Float64`, and promotion happens with the multiplication. Similarly, `Rational` literals are less type disruptive than `Float64` literals, but more disruptive than `Int`s:

```
julia> h(x) = 2//1 * x
h (generic function with 1 method)

julia> h(1//2)
1//1

julia> h(1/2)
1.0

julia> h(1)
2//1
```

Thus, use `Int` literals when possible, with `Rational{Int}` for literal non-integer numbers, in order to make it easier to use your code.

Chapter 39

Frequently Asked Questions

39.1 General

Is Julia named after someone or something?

No.

Why don't you compile Matlab/Python/R/... code to Julia?

Since many people are familiar with the syntax of other dynamic languages, and lots of code has already been written in those languages, it is natural to wonder why we didn't just plug a Matlab or Python front-end into a Julia back-end (or "transpile" code to Julia) in order to get all the performance benefits of Julia without requiring programmers to learn a new language. Simple, right?

The basic issue is that there is *nothing special about Julia's compiler*: we use a commonplace compiler (LLVM) with no "secret sauce" that other language developers don't know about. Indeed, Julia's compiler is in many ways much simpler than those of other dynamic languages (e.g. PyPy or LuaJIT). Julia's performance advantage derives almost entirely from its front-end: its language semantics allow a well-written Julia program to *give more opportunities to the compiler* to generate efficient code and memory layouts. If you tried to compile Matlab or Python code to Julia, our compiler would be limited by the semantics of Matlab or Python to producing code no better than that of existing compilers for those languages (and probably worse). The key role of semantics is also why several existing Python compilers (like Numba and Pythran) only attempt to optimize a small subset of the language (e.g. operations on Numpy arrays and scalars), and for this subset they are already doing at least as well as we could for the same semantics. The people working on those projects are incredibly smart and have accomplished amazing things, but retrofitting a compiler onto a language that was designed to be interpreted is a very difficult problem.

Julia's advantage is that good performance is not limited to a small subset of "built-in" types and operations, and one can write high-level type-generic code that works on arbitrary user-defined types while remaining fast and memory-efficient. Types in languages like Python simply don't provide enough information to the compiler for similar capabilities, so as soon as you used those languages as a Julia front-end you would be stuck.

For similar reasons, automated translation to Julia would also typically generate unreadable, slow, non-idiomatic code that would not be a good starting point for a native Julia port from another language.

On the other hand, language *interoperability* is extremely useful: we want to exploit existing high-quality code in other languages from Julia (and vice versa)! The best way to enable this is not a transpiler, but

rather via easy inter-language calling facilities. We have worked hard on this, from the built-in `ccall` intrinsic (to call C and Fortran libraries) to JuliaInterop packages that connect Julia to Python, Matlab, C++, and more.

39.2 Public API

How does Julia define its public API?

Julia's public API is the behavior described in documentation of public bindings from Base and the standard libraries. Functions, types, and constants are not part of the public API if they are not public, even if they have docstrings or are described in the documentation. Further, only the documented behavior of public bindings is part of the public API. Undocumented behavior of public bindings is internal.

Public bindings are those marked with either `public foo` or `export foo`.

In other words:

- Documented behavior of public bindings is part of the public API.
- Undocumented behavior of public bindings is not part of the public API.
- Documented behavior of private bindings is not part of the public API.
- Undocumented behavior of private bindings is not part of the public API.

You can get a complete list of the public bindings from a module with `names(MyModule)`.

Package authors are encouraged to define their public API similarly.

Anything in Julia's Public API is covered by SemVer and therefore will not be removed or receive meaningful breaking changes before Julia 2.0.

There is a useful undocumented function/type/constant. Can I use it?

Updating Julia may break your code if you use non-public API. If the code is self-contained, it may be a good idea to copy it into your project. If you want to rely on a complex non-public API, especially when using it from a stable package, it is a good idea to open an issue or pull request to start a discussion for turning it into a public API. However, we do not discourage the attempt to create packages that expose stable public interfaces while relying on non-public implementation details of Julia and buffering the differences across different Julia versions.

The documentation is not accurate enough. Can I rely on the existing behavior?

Please open an issue or pull request to start a discussion for turning the existing behavior into a public API.

39.3 Sessions and the REPL

How do I delete an object in memory?

Julia does not have an analog of MATLAB's `clear` function; once a name is defined in a Julia session (technically, in module `Main`), it is always present.

If memory usage is your concern, you can always replace objects with ones that consume less memory. For example, if `A` is a gigabyte-sized array that you no longer need, you can free the memory with `A = nothing`. The memory will be released the next time the garbage collector runs; you can force this to happen with `GC.gc()`. Moreover, an attempt to use `A` will likely result in an error, because most methods are not defined on type `Nothing`.

39.4 Scripting

How do I check if the current file is being run as the main script?

When a file is run as the main script using `julia file.jl` one might want to activate extra functionality like command line argument handling. A way to determine that a file is run in this fashion is to check if `abspath(PROGRAM_FILE) == @__FILE__` is `true`.

However, it is recommended to not write files that double as a script and as an importable library. If one needs functionality both available as a library and a script, it is better to write is as a library, then import the functionality into a distinct script.

How do I catch CTRL-C in a script?

Running a Julia script using `julia file.jl` does not throw `InterruptException` when you try to terminate it with CTRL-C (SIGINT). To run a certain code before terminating a Julia script, which may or may not be caused by CTRL-C, use `atexit`. Alternatively, you can use `julia -e 'include(popfirst!(ARGS))' file.jl` to execute a script while being able to catch `InterruptException` in the `try` block. Note that with this strategy `PROGRAM_FILE` will not be set.

How do I pass options to `julia` using `#!/usr/bin/env`?

Passing options to `julia` in a so-called shebang line, as in `#!/usr/bin/env julia --startup-file=no`, will not work on many platforms (BSD, macOS, Linux) where the kernel, unlike the shell, does not split arguments at space characters. The option `env -S`, which splits a single argument string into multiple arguments at spaces, similar to a shell, offers a simple workaround:

```
#!/usr/bin/env -S julia --color=yes --startup-file=no
@show ARGS  # put any Julia code here
```

Note

Option `env -S` appeared in FreeBSD 6.0 (2005), macOS Sierra (2016) and GNU/Linux coreutils 8.30 (2018).

Why doesn't `run` support * or pipes for scripting external programs?

Julia's `run` function launches external programs *directly*, without invoking an operating-system shell (unlike the `system("...")` function in other languages like Python, R, or C). That means that `run` does not perform wildcard expansion of * ("globbing"), nor does it interpret shell pipelines like | or >.

You can still do globbing and pipelines using Julia features, however. For example, the built-in `pipeline` function allows you to chain external programs and files, similar to shell pipes, and the Glob.jl package implements POSIX-compatible globbing.

You can, of course, run programs through the shell by explicitly passing a shell and a command string to `run`, e.g. `run(`sh -c "ls > files.txt"`)` to use the Unix Bourne shell, but you should generally prefer pure-Julia scripting like `run(pipeline(`ls`, "files.txt"))`. The reason why we avoid the shell by default is that shelling out sucks: launching processes via the shell is slow, fragile to quoting of special characters, has poor error handling, and is problematic for portability. (The Python developers came to a similar conclusion.)

39.5 Variables and Assignments

Why am I getting `UndefVarError` from a simple loop?

You might have something like:

```
x = 0
while x < 10
    x += 1
end
```

and notice that it works fine in an interactive environment (like the Julia REPL), but gives `UndefVarError: `x` not defined` when you try to run it in script or other file. What is going on is that Julia generally requires you to **be explicit about assigning to global variables in a local scope**.

Here, `x` is a global variable, `while` defines a local scope, and `x += 1` is an assignment to a global in that local scope.

As mentioned above, Julia (version 1.5 or later) allows you to omit the `global` keyword for code in the REPL (and many other interactive environments), to simplify exploration (e.g. copy-pasting code from a function to run interactively). However, once you move to code in files, Julia requires a more disciplined approach to global variables. You have least three options:

1. Put the code into a function (so that `x` is a *local* variable in a function). In general, it is good software engineering to use functions rather than global scripts (search online for "why global variables bad" to see many explanations). In Julia, global variables are also slow.
2. Wrap the code in a `let` block. (This makes `x` a local variable within the `let ... end` statement, again eliminating the need for `global`).
3. Explicitly mark `x` as `global` inside the local scope before assigning to it, e.g. write `global x += 1`.

More explanation can be found in the manual section on soft scope.

39.6 Functions

I passed an argument `x` to a function, modified it inside that function, but on the outside, the variable `x` is still unchanged. Why?

Suppose you call a function like this:

```
julia> x = 10
10

julia> function change_value!(y)
           y = 17
       end
change_value! (generic function with 1 method)

julia> change_value!(x)
17

julia> x # x is unchanged!
10
```

In Julia, the binding of a variable x cannot be changed by passing x as an argument to a function. When calling `change_value!(x)` in the above example, y is a newly created variable, bound initially to the value of x, i.e. `10`; then y is rebound to the constant 17, while the variable x of the outer scope is left untouched.

However, if x is bound to an object of type `Array` (or any other *mutable* type). From within the function, you cannot "unbind" x from this Array, but you *can* change its content. For example:

```
julia> x = [1,2,3]
3-element Vector{Int64}:
 1
 2
 3

julia> function change_array!(A)
           A[1] = 5
       end
change_array! (generic function with 1 method)

julia> change_array!(x)
5

julia> x
3-element Vector{Int64}:
 5
 2
 3
```

Here we created a function `change_array!`, that assigns 5 to the first element of the passed array (bound to x at the call site, and bound to `A` within the function). Notice that, after the function call, x is still bound to the same array, but the content of that array changed: the variables `A` and x were distinct bindings referring to the same mutable `Array` object.

Can I use `using` or `import` inside a function?

No, you are not allowed to have a `using` or `import` statement inside a function. If you want to import a module but only use its bindings inside a specific function or set of functions, you have two options:

1. Use `import`:

```
import Foo
function bar(...)
    # ... refer to Foo bindings via Foo.baz ...
end
```

This loads the module Foo and defines a variable Foo that refers to the module, but does not import any of the other bindings from the module into the current namespace. You refer to the Foo bindings by their qualified names `Foo.bar` etc.

2. Wrap your function in a module:

```
module Bar
export bar
using Foo
function bar(...)
    # ... refer to Foo.baz as simply baz ....
end
end
using Bar
```

This imports all the bindings from Foo, but only inside the module `Bar`.

What does the `...` operator do?

The two uses of the `...` operator: slurping and splatting

Many newcomers to Julia find the use of `...` operator confusing. Part of what makes the `...` operator confusing is that it means two different things depending on context.

`...` combines many arguments into one argument in function definitions

In the context of function definitions, the `...` operator is used to combine many different arguments into a single argument. This use of `...` for combining many different arguments into a single argument is called slurping:

```
julia> function printargs(args...)
           println(typeof(args))
           for (i, arg) in enumerate(args)
               println("Arg #$i = $arg")
           end
       end
printargs (generic function with 1 method)

julia> printargs(1, 2, 3)
Tuple{Int64, Int64, Int64}
Arg #1 = 1
Arg #2 = 2
Arg #3 = 3
```

If Julia were a language that made more liberal use of ASCII characters, the slurping operator might have been written as `<-...` instead of `....`

... splits one argument into many different arguments in function calls

In contrast to the use of the ... operator to denote slurping many different arguments into one argument when defining a function, the ... operator is also used to cause a single function argument to be split apart into many different arguments when used in the context of a function call. This use of ... is called splatting:

```
julia> function threeargs(a, b, c)
           println("a = $a::$(typeof(a))")
           println("b = $b::$(typeof(b))")
           println("c = $c::$(typeof(c))")
       end
threeargs (generic function with 1 method)

julia> x = [1, 2, 3]
3-element Vector{Int64}:
 1
 2
 3

julia> threeargs(x...)
a = 1::Int64
b = 2::Int64
c = 3::Int64
```

If Julia were a language that made more liberal use of ASCII characters, the splatting operator might have been written as ...-> instead of ....

What is the return value of an assignment?

The operator = always returns the right-hand side, therefore:

```
julia> function threeint()
           x::Int = 3.0
           x # returns variable x
       end
threeint (generic function with 1 method)

julia> function threefloat()
           x::Int = 3.0 # returns 3.0
       end
threefloat (generic function with 1 method)

julia> threeint()
3

julia> threefloat()
3.0
```

and similarly:

```
julia> function twothreetup()
           x, y = [2, 3] # assigns 2 to x and 3 to y
           x, y # returns a tuple
       end
twothreetup (generic function with 1 method)

julia> function twothreearr()
           x, y = [2, 3] # returns an array
       end
twothreearr (generic function with 1 method)

julia> twothreetup()
(2, 3)

julia> twothreearr()
2-element Vector{Int64}:
 2
 3
```

Is a function that ends with ! allowed to allocate?

Yes! A function name ending with ! indicates that the function mutates at least one of its arguments (typically the first argument). However, it may still allocate a scratch space to expedite computation or produce that result.

39.7 Types, type declarations, and constructors

What does "type-stable" mean?

It means that the type of the output is predictable from the types of the inputs. In particular, it means that the type of the output cannot vary depending on the *values* of the inputs. The following code is *not* type-stable:

```
julia> function unstable(flag::Bool)
           if flag
               return 1
           else
               return 1.0
           end
       end
unstable (generic function with 1 method)
```

It returns either an `Int` or a `Float64` depending on the value of its argument. Since Julia can't predict the return type of this function at compile-time, any computation that uses it must be able to cope with values of both types, which makes it hard to produce fast machine code.

Why does Julia give a `DomainError` for certain seemingly-sensible operations?

Certain operations make mathematical sense but result in errors:

```
julia> sqrt(-2.0)
```

```
ERROR: DomainError with -2.0:
sqrt was called with a negative real argument but will only return a complex result if called
↪  with a complex argument. Try sqrt(Complex(x)).
Stacktrace:
[...]
```

This behavior is an inconvenient consequence of the requirement for type-stability. In the case of `sqrt`, most users want `sqrt(2.0)` to give a real number, and would be unhappy if it produced the complex number `1.4142135623730951 + 0.0im`. One could write the `sqrt` function to switch to a complex-valued output only when passed a negative number (which is what `sqrt` does in some other languages), but then the result would not be type-stable and the `sqrt` function would have poor performance.

In these and other cases, you can get the result you want by choosing an *input type* that conveys your willingness to accept an *output type* in which the result can be represented:

```
julia> sqrt(-2.0+0im)
0.0 + 1.4142135623730951im
```

How can I constrain or compute type parameters?

The parameters of a parametric type can hold either types or bits values, and the type itself chooses how it makes use of these parameters. For example, `Array{Float64, 2}` is parameterized by the type `Float64` to express its element type and the integer value 2 to express its number of dimensions. When defining your own parametric type, you can use subtype constraints to declare that a certain parameter must be a subtype (`<:`) of some abstract type or a previous type parameter. There is not, however, a dedicated syntax to declare that a parameter must be a *value* of a given type — that is, you cannot directly declare that a dimensionality-like parameter `isa Int` within the `struct` definition, for example. Similarly, you cannot do computations (including simple things like addition or subtraction) on type parameters. Instead, these sorts of constraints and relationships may be expressed through additional type parameters that are computed and enforced within the type's constructors.

As an example, consider

```
struct ConstrainedType{T,N,N+1} # NOTE: INVALID SYNTAX
    A::Array{T,N}
    B::Array{T,N+1}
end
```

where the user would like to enforce that the third type parameter is always the second plus one. This can be implemented with an explicit type parameter that is checked by an inner constructor method (where it can be combined with other checks):

```
struct ConstrainedType{T,N,M}
    A::Array{T,N}
    B::Array{T,M}
    function ConstrainedType(A::Array{T,N}, B::Array{T,M}) where {T,N,M}
        N + 1 == M || throw(ArgumentError("second argument should have one more axis" ))
        new{T,N,M}(A, B)
    end
end
```

This check is usually *costless*, as the compiler can elide the check for valid concrete types. If the second argument is also computed, it may be advantageous to provide an outer constructor method that performs this calculation:

```
ConstrainedType(A) = ConstrainedType(A, compute_B(A))
```

Why does Julia use native machine integer arithmetic?

Julia uses machine arithmetic for integer computations. This means that the range of `Int` values is bounded and wraps around at either end so that adding, subtracting and multiplying integers can overflow or underflow, leading to some results that can be unsettling at first:

```
julia> x = typemax(Int)
9223372036854775807

julia> y = x+1
-9223372036854775808

julia> z = -y
-9223372036854775808

julia> 2*z
0
```

Clearly, this is far from the way mathematical integers behave, and you might think it less than ideal for a high-level programming language to expose this to the user. For numerical work where efficiency and transparency are at a premium, however, the alternatives are worse.

One alternative to consider would be to check each integer operation for overflow and promote results to bigger integer types such as `Int128` or `BigInt` in the case of overflow. Unfortunately, this introduces major overhead on every integer operation (think incrementing a loop counter) – it requires emitting code to perform run-time overflow checks after arithmetic instructions and branches to handle potential overflows. Worse still, this would cause every computation involving integers to be type-unstable. As we mentioned above, type-stability is crucial for effective generation of efficient code. If you can't count on the results of integer operations being integers, it's impossible to generate fast, simple code the way C and Fortran compilers do.

A variation on this approach, which avoids the appearance of type instability is to merge the `Int` and `BigInt` types into a single hybrid integer type, that internally changes representation when a result no longer fits into the size of a machine integer. While this superficially avoids type-instability at the level of Julia code, it just sweeps the problem under the rug by foisting all of the same difficulties onto the C code implementing this hybrid integer type. This approach *can* be made to work and can even be made quite fast in many cases, but has several drawbacks. One problem is that the in-memory representation of integers and arrays of integers no longer match the natural representation used by C, Fortran and other languages with native machine integers. Thus, to interoperate with those languages, we would ultimately need to introduce native integer types anyway. Any unbounded representation of integers cannot have a fixed number of bits, and thus cannot be stored inline in an array with fixed-size slots – large integer values will always require separate heap-allocated storage. And of course, no matter how clever a hybrid integer implementation one uses, there are always performance traps – situations where performance degrades unexpectedly. Complex representation, lack of interoperability with C and Fortran, the inability to represent

integer arrays without additional heap storage, and unpredictable performance characteristics make even the cleverest hybrid integer implementations a poor choice for high-performance numerical work.

An alternative to using hybrid integers or promoting to BigInts is to use saturating integer arithmetic, where adding to the largest integer value leaves it unchanged and likewise for subtracting from the smallest integer value. This is precisely what Matlab™ does:

```
>> int64(9223372036854775807)

ans =

  9223372036854775807

>> int64(9223372036854775807) + 1

ans =

  9223372036854775807

>> int64(-9223372036854775808)

ans =

 -9223372036854775808

>> int64(-9223372036854775808) - 1

ans =

 -9223372036854775808
```

At first blush, this seems reasonable enough since 9223372036854775807 is much closer to 9223372036854775808 than -9223372036854775808 is and integers are still represented with a fixed size in a natural way that is compatible with C and Fortran. Saturated integer arithmetic, however, is deeply problematic. The first and most obvious issue is that this is not the way machine integer arithmetic works, so implementing saturated operations requires emitting instructions after each machine integer operation to check for underflow or overflow and replace the result with `typemin(Int)` or `typemax(Int)` as appropriate. This expands each integer operation from a single, fast instruction into a few instructions. But it gets worse – saturating integer arithmetic isn't associative. Consider this Matlab computation:

```
>> n = int64(2)^62
4611686018427387904

>> n + (n - 1)
9223372036854775807

>> (n + n) - 1
9223372036854775806
```

This makes it hard to write many basic integer algorithms since a lot of common techniques depend on the fact that machine addition with overflow *is* associative. Consider finding the midpoint between integer values `lo` and `hi` in Julia using the expression `(lo + hi) >>> 1`:

```
julia> n = 2^62
4611686018427387904

julia> (n + 2n) >>> 1
6917529027641081856
```

See? No problem. That's the correct midpoint between 2^62 and 2^63, despite the fact that `n + 2n` is -4611686018427387904. Now try it in Matlab:

```
>> (n + 2*n)/2

ans =

  4611686018427387904
```

Oops. Adding a >>> operator to Matlab wouldn't help, because saturation that occurs when adding `n` and `2n` has already destroyed the information necessary to compute the correct midpoint.

Not only is lack of associativity unfortunate for programmers who cannot rely it for techniques like this, but it also defeats almost anything compilers might want to do to optimize integer arithmetic. For example, since Julia integers use normal machine integer arithmetic, LLVM is free to aggressively optimize simple little functions like `f(k) = 5k-1`. The machine code for this function is just this:

```
julia> code_native(f, Tuple{Int})
  .text
Filename: none
  pushq %rbp
  movq  %rsp, %rbp
Source line: 1
  leaq  -1(%rdi,%rdi,4), %rax
  popq  %rbp
  retq
  nopl  (%rax,%rax)
```

The actual body of the function is a single `leaq` instruction, which computes the integer multiply and add at once. This is even more beneficial when `f` gets inlined into another function:

```
julia> function g(k, n)
           for i = 1:n
               k = f(k)
           end
           return k
       end
g (generic function with 1 methods)

julia> code_native(g, Tuple{Int,Int})
  .text
Filename: none
  pushq %rbp
  movq  %rsp, %rbp
Source line: 2
```

```
  testq %rsi, %rsi
  jle L26
  nopl  (%rax)
Source line: 3
L16:
  leaq  -1(%rdi,%rdi,4), %rdi
Source line: 2
  decq  %rsi
  jne L16
Source line: 5
L26:
  movq  %rdi, %rax
  popq  %rbp
  retq
  nop
```

Since the call to `f` gets inlined, the loop body ends up being just a single `leaq` instruction. Next, consider what happens if we make the number of loop iterations fixed:

```
julia> function g(k)
           for i = 1:10
               k = f(k)
           end
           return k
       end
g (generic function with 2 methods)

julia> code_native(g,(Int,))
  .text
Filename: none
  pushq %rbp
  movq  %rsp, %rbp
Source line: 3
  imulq $9765625, %rdi, %rax    # imm = 0x9502F9
  addq  $-2441406, %rax         # imm = 0xFFDABF42
Source line: 5
  popq  %rbp
  retq
  nopw  %cs:(%rax,%rax)
```

Because the compiler knows that integer addition and multiplication are associative and that multiplication distributes over addition – neither of which is true of saturating arithmetic – it can optimize the entire loop down to just a multiply and an add. Saturated arithmetic completely defeats this kind of optimization since associativity and distributivity can fail at each loop iteration, causing different outcomes depending on which iteration the failure occurs in. The compiler can unroll the loop, but it cannot algebraically reduce multiple operations into fewer equivalent operations.

The most reasonable alternative to having integer arithmetic silently overflow is to do checked arithmetic everywhere, raising errors when adds, subtracts, and multiplies overflow, producing values that are not value-correct. In this blog post, Dan Luu analyzes this and finds that rather than the trivial cost that this approach should in theory have, it ends up having a substantial cost due to compilers (LLVM and GCC) not

gracefully optimizing around the added overflow checks. If this improves in the future, we could consider defaulting to checked integer arithmetic in Julia, but for now, we have to live with the possibility of overflow.

In the meantime, overflow-safe integer operations can be achieved through the use of external libraries such as SaferIntegers.jl. Note that, as stated previously, the use of these libraries significantly increases the execution time of code using the checked integer types. However, for limited usage, this is far less of an issue than if it were used for all integer operations. You can follow the status of the discussion here.

What are the possible causes of an `UndefVarError` during remote execution?

As the error states, an immediate cause of an `UndefVarError` on a remote node is that a binding by that name does not exist. Let us explore some of the possible causes.

```
julia> module Foo
           foo() = remotecall_fetch(x->x, 2, "Hello")
       end

julia> Foo.foo()
ERROR: On worker 2:
UndefVarError: `Foo` not defined in `Main`
Stacktrace:
[...]
```

The closure `x->x` carries a reference to `Foo`, and since `Foo` is unavailable on node 2, an `UndefVarError` is thrown.

Globals under modules other than `Main` are not serialized by value to the remote node. Only a reference is sent. Functions which create global bindings (except under `Main`) may cause an `UndefVarError` to be thrown later.

```
julia> @everywhere module Foo
           function foo()
               global gvar = "Hello"
               remotecall_fetch(()->gvar, 2)
           end
       end

julia> Foo.foo()
ERROR: On worker 2:
UndefVarError: `gvar` not defined in `Main.Foo`
Stacktrace:
[...]
```

In the above example, `@everywhere module Foo` defined `Foo` on all nodes. However the call to `Foo.foo()` created a new global binding `gvar` on the local node, but this was not found on node 2 resulting in an `UndefVarError` error.

Note that this does not apply to globals created under module `Main`. Globals under module `Main` are serialized and new bindings created under `Main` on the remote node.

```
julia> gvar_self = "Node1"
```

```
"Node1"

julia> remotecall_fetch(()->gvar_self, 2)
"Node1"

julia> remotecall_fetch(varinfo, 2)
name          size summary
_________ ________ _______
Base               Module
Core               Module
Main               Module
gvar_self 13 bytes String
```

This does not apply to `function` or `struct` declarations. However, anonymous functions bound to global variables are serialized as can be seen below.

```
julia> bar() = 1
bar (generic function with 1 method)

julia> remotecall_fetch(bar, 2)
ERROR: On worker 2:
UndefVarError: `#bar` not defined in `Main`
[...]

julia> anon_bar  = ()->1
(::#21) (generic function with 1 method)

julia> remotecall_fetch(anon_bar, 2)
1
```

39.8 Troubleshooting "method not matched": parametric type invariance and `MethodErrors`

Why doesn't it work to declare `foo(bar::Vector{Real}) = 42` and then call `foo([1])`?

As you'll see if you try this, the result is a `MethodError`:

```
julia> foo(x::Vector{Real}) = 42
foo (generic function with 1 method)

julia> foo([1])
ERROR: MethodError: no method matching foo(::Vector{Int64})
The function `foo` exists, but no method is defined for this combination of argument types.

Closest candidates are:
  foo(!Matched::Vector{Real})
   @ Main none:1

Stacktrace:
[...]
```

This is because `Vector{Real}` is not a supertype of `Vector{Int}`! You can solve this problem with something like `foo(bar::Vector{T}) where {T<:Real}` (or the short form `foo(bar::Vector{<:Real})` if the static parameter T is not needed in the body of the function). The T is a wild card: you first specify that it must be a subtype of Real, then specify the function takes a Vector of with elements of that type.

This same issue goes for any composite type `Comp`, not just `Vector`. If `Comp` has a parameter declared of type `Y`, then another type `Comp2` with a parameter of type `X<:Y` is not a subtype of `Comp`. This is type-invariance (by contrast, Tuple is type-covariant in its parameters). See Parametric Composite Types for more explanation of these.

Why does Julia use * for string concatenation? Why not + or something else?

The main argument against + is that string concatenation is not commutative, while + is generally used as a commutative operator. While the Julia community recognizes that other languages use different operators and * may be unfamiliar for some users, it communicates certain algebraic properties.

Note that you can also use `string(...)` to concatenate strings (and other values converted to strings); similarly, `repeat` can be used instead of ^ to repeat strings. The interpolation syntax is also useful for constructing strings.

39.9 Packages and Modules

What is the difference between "using" and "import"?

There are several differences between `using` and `import` (see the Modules section), but there is an important difference that may not seem intuitive at first glance, and on the surface (i.e. syntax-wise) it may seem very minor. When loading modules with `using`, you need to say `function Foo.bar(...` to extend module Foo's function `bar` with a new method, but with `import Foo.bar`, you only need to say `function bar(...` and it automatically extends module Foo's function `bar`.

The reason this is important enough to have been given separate syntax is that you don't want to accidentally extend a function that you didn't know existed, because that could easily cause a bug. This is most likely to happen with a method that takes a common type like a string or integer, because both you and the other module could define a method to handle such a common type. If you use `import`, then you'll replace the other module's implementation of `bar(s::AbstractString)` with your new implementation, which could easily do something completely different (and break all/many future usages of the other functions in module Foo that depend on calling bar).

39.10 Nothingness and missing values

How does "null", "nothingness" or "missingness" work in Julia?

Unlike many languages (for example, C and Java), Julia objects cannot be "null" by default. When a reference (variable, object field, or array element) is uninitialized, accessing it will immediately throw an error. This situation can be detected using the `isdefined` or `isassigned` functions.

Some functions are used only for their side effects, and do not need to return a value. In these cases, the convention is to return the value `nothing`, which is just a singleton object of type `Nothing`. This is an ordinary type with no fields; there is nothing special about it except for this convention, and that the REPL does not print anything for it. Some language constructs that would not otherwise have a value also yield `nothing`, for example `if false; end`.

For situations where a value x of type T exists only sometimes, the `Union{T, Nothing}` type can be used for function arguments, object fields and array element types as the equivalent of `Nullable`, `Option` or `Maybe` in other languages. If the value itself can be `nothing` (notably, when T is `Any`), the `Union{Some{T}, Nothing}` type is more appropriate since `x == nothing` then indicates the absence of a value, and `x == Some(nothing)` indicates the presence of a value equal to `nothing`. The `something` function allows unwrapping `Some` objects and using a default value instead of `nothing` arguments. Note that the compiler is able to generate efficient code when working with `Union{T, Nothing}` arguments or fields.

To represent missing data in the statistical sense (`NA` in R or `NULL` in SQL), use the `missing` object. See the `Missing Values` section for more details.

In some languages, the empty tuple (`()`) is considered the canonical form of nothingness. However, in julia it is best thought of as just a regular tuple that happens to contain zero values.

The empty (or "bottom") type, written as `Union{}` (an empty union type), is a type with no values and no subtypes (except itself). You will generally not need to use this type.

39.11 Memory

Why does `x += y` allocate memory when `x` and `y` are arrays?

In Julia, `x += y` gets replaced during lowering by `x = x + y`. For arrays, this has the consequence that, rather than storing the result in the same location in memory as `x`, it allocates a new array to store the result. If you prefer to mutate `x`, use `x .+= y` to update each element individually.

While this behavior might surprise some, the choice is deliberate. The main reason is the presence of immutable objects within Julia, which cannot change their value once created. Indeed, a number is an immutable object; the statements `x = 5; x += 1` do not modify the meaning of 5, they modify the value bound to `x`. For an immutable, the only way to change the value is to reassign it.

To amplify a bit further, consider the following function:

```
function power_by_squaring(x, n::Int)
    ispow2(n) || error("This implementation only works for powers of 2")
    while n >= 2
        x *= x
        n >>= 1
    end
    x
end
```

After a call like `x = 5; y = power_by_squaring(x, 4)`, you would get the expected result: `x == 5 && y == 625`. However, now suppose that *=, when used with matrices, instead mutated the left hand side. There would be two problems:

- For general square matrices, `A = A*B` cannot be implemented without temporary storage: `A[1,1]` gets computed and stored on the left hand side before you're done using it on the right hand side.
- Suppose you were willing to allocate a temporary for the computation (which would eliminate most of the point of making *= work in-place); if you took advantage of the mutability of `x`, then this function would behave differently for mutable vs. immutable inputs. In particular, for immutable `x`, after the call you'd have (in general) `y != x`, but for mutable `x` you'd have `y == x`.

Because supporting generic programming is deemed more important than potential performance optimizations that can be achieved by other means (e.g., using broadcasting or explicit loops), operators like += and *= work by rebinding new values.

39.12 Asynchronous IO and concurrent synchronous writes

Why do concurrent writes to the same stream result in inter-mixed output?

While the streaming I/O API is synchronous, the underlying implementation is fully asynchronous.

Consider the printed output from the following:

```
julia> @sync for i in 1:3
           Threads.@spawn write(stdout, string(i), " Foo ", " Bar ")
       end
123 Foo  Foo  Foo  Bar  Bar  Bar
```

This is happening because, while the `write` call is synchronous, the writing of each argument yields to other tasks while waiting for that part of the I/O to complete.

`print` and `println` "lock" the stream during a call. Consequently changing `write` to `println` in the above example results in:

```
julia> @sync for i in 1:3
           Threads.@spawn println(stdout, string(i), " Foo ", " Bar ")
       end
1 Foo  Bar
2 Foo  Bar
3 Foo  Bar
```

You can lock your writes with a `ReentrantLock` like this:

```
julia> l = ReentrantLock();

julia> @sync for i in 1:3
           Threads.@spawn begin
               lock(l)
               try
                   write(stdout, string(i), " Foo ", " Bar ")
               finally
                   unlock(l)
               end
           end
       end
1 Foo  Bar 2 Foo  Bar 3 Foo  Bar
```

39.13 Arrays

What are the differences between zero-dimensional arrays and scalars?

Zero-dimensional arrays are arrays of the form `Array{T,0}`. They behave similar to scalars, but there are important differences. They deserve a special mention because they are a special case which makes

logical sense given the generic definition of arrays, but might be a bit unintuitive at first. The following line defines a zero-dimensional array:

```
julia> A = zeros()
0-dimensional Array{Float64,0}:
0.0
```

In this example, `A` is a mutable container that contains one element, which can be set by `A[] = 1.0` and retrieved with `A[]`. All zero-dimensional arrays have the same size (`size(A) == ()`), and length (`length(A) == 1`). In particular, zero-dimensional arrays are not empty. If you find this unintuitive, here are some ideas that might help to understand Julia's definition.

- Zero-dimensional arrays are the "point" to vector's "line" and matrix's "plane". Just as a line has no area (but still represents a set of things), a point has no length or any dimensions at all (but still represents a thing).
- We define `prod(()) to be 1`, and the total number of elements in an array is the product of the size. The size of a zero-dimensional array is `()`, and therefore its length is `1`.
- Zero-dimensional arrays don't natively have any dimensions into which you index – they're just `A[]`. We can apply the same "trailing one" rule for them as for all other array dimensionalities, so you can indeed index them as `A[1]`, `A[1,1]`, etc; see Omitted and extra indices.

It is also important to understand the differences to ordinary scalars. Scalars are not mutable containers (even though they are iterable and define things like `length`, `getindex`, *e.g.* `1[] == 1`). In particular, if `x = 0.0` is defined as a scalar, it is an error to attempt to change its value via `x[] = 1.0`. A scalar `x` can be converted into a zero-dimensional array containing it via `fill(x)`, and conversely, a zero-dimensional array `a` can be converted to the contained scalar via `a[]`. Another difference is that a scalar can participate in linear algebra operations such as `2 * rand(2,2)`, but the analogous operation with a zero-dimensional array `fill(2) * rand(2,2)` is an error.

Why are my Julia benchmarks for linear algebra operations different from other languages?

You may find that simple benchmarks of linear algebra building blocks like

```
using BenchmarkTools
A = randn(1000, 1000)
B = randn(1000, 1000)
@btime $A \ $B
@btime $A * $B
```

can be different when compared to other languages like Matlab or R.

Since operations like this are very thin wrappers over the relevant BLAS functions, the reason for the discrepancy is very likely to be

1. the BLAS library each language is using,

2. the number of concurrent threads.

Julia compiles and uses its own copy of OpenBLAS, with threads currently capped at 8 (or the number of your cores).

Modifying OpenBLAS settings or compiling Julia with a different BLAS library, eg Intel MKL, may provide performance improvements. You can use MKL.jl, a package that makes Julia's linear algebra use Intel MKL BLAS and LAPACK instead of OpenBLAS, or search the discussion forum for suggestions on how to set this up manually. Note that Intel MKL cannot be bundled with Julia, as it is not open source.

39.14 Computing cluster

How do I manage precompilation caches in distributed file systems?

When using Julia in high-performance computing (HPC) facilities with shared filesystems, it is recommended to use a shared depot (via the `JULIA_DEPOT_PATH` environment variable). Since Julia v1.10, multiple Julia processes on functionally similar workers and using the same depot will coordinate via pidfile locks to only spend effort precompiling on one process while the others wait. The precompilation process will indicate when the process is precompiling or waiting for another that is precompiling. If non-interactive the messages are via `@debug`.

However, due to caching of binary code, the cache rejection since v1.9 is more strict and users may need to set the `JULIA_CPU_TARGET` environment variable appropriately to get a single cache that is usable throughout the HPC environment.

39.15 Julia Releases

Do I want to use the Stable, LTS, or nightly version of Julia?

The Stable version of Julia is the latest released version of Julia, this is the version most people will want to run. It has the latest features, including improved performance. The Stable version of Julia is versioned according to SemVer as v1.x.y. A new minor release of Julia corresponding to a new Stable version is made approximately every 4-5 months after a few weeks of testing as a release candidate. Unlike the LTS version the Stable version will not normally receive bugfixes after another Stable version of Julia has been released. However, upgrading to the next Stable release will always be possible as each release of Julia v1.x will continue to run code written for earlier versions.

You may prefer the LTS (Long Term Support) version of Julia if you are looking for a very stable code base. The current LTS version of Julia is versioned according to SemVer as v1.6.x; this branch will continue to receive bugfixes until a new LTS branch is chosen, at which point the v1.6.x series will no longer received regular bug fixes and all but the most conservative users will be advised to upgrade to the new LTS version series. As a package developer, you may prefer to develop for the LTS version, to maximize the number of users who can use your package. As per SemVer, code written for v1.0 will continue to work for all future LTS and Stable versions. In general, even if targeting the LTS, one can develop and run code in the latest Stable version, to take advantage of the improved performance; so long as one avoids using new features (such as added library functions or new methods).

You may prefer the nightly version of Julia if you want to take advantage of the latest updates to the language, and don't mind if the version available today occasionally doesn't actually work. As the name implies, releases to the nightly version are made roughly every night (depending on build infrastructure stability). In general nightly released are fairly safe to use—your code will not catch on fire. However, they

may be occasional regressions and or issues that will not be found until more thorough pre-release testing. You may wish to test against the nightly version to ensure that such regressions that affect your use case are caught before a release is made.

Finally, you may also consider building Julia from source for yourself. This option is mainly for those individuals who are comfortable at the command line, or interested in learning. If this describes you, you may also be interested in reading our guidelines for contributing.

The `juliaup` install manager has pre-defined channels named `release` and `lts` for the latest stable release and the current LTS release, as well as version-specific channels.

How can I transfer the list of installed packages after updating my version of Julia?

Each minor version of julia has its own default environment. As a result, upon installing a new minor version of Julia, the packages you added using the previous minor version will not be available by default. The environment for a given julia version is defined by the files `Project.toml` and `Manifest.toml` in a folder matching the version number in `.julia/environments/`, for instance, `.julia/environments/v1.3`.

If you install a new minor version of Julia, say `1.4`, and want to use in its default environment the same packages as in a previous version (e.g. `1.3`), you can copy the contents of the file `Project.toml` from the `1.3` folder to `1.4`. Then, in a session of the new Julia version, enter the "package management mode" by typing the key `]`, and run the command `instantiate`.

This operation will resolve a set of feasible packages from the copied file that are compatible with the target Julia version, and will install or update them if suitable. If you want to reproduce not only the set of packages, but also the versions you were using in the previous Julia version, you should also copy the `Manifest.toml` file before running the Pkg command `instantiate`. However, note that packages may define compatibility constraints that may be affected by changing the version of Julia, so the exact set of versions you had in `1.3` may not work for `1.4`.

Chapter 40

Noteworthy Differences from other Languages

40.1 Noteworthy differences from MATLAB

Although MATLAB users may find Julia's syntax familiar, Julia is not a MATLAB clone. There are major syntactic and functional differences. The following are some noteworthy differences that may trip up Julia users accustomed to MATLAB:

- Julia arrays are indexed with square brackets, `A[i,j]`.
- Julia arrays are not copied when assigned to another variable. After `A = B`, changing elements of `B` will modify `A` as well. To avoid this, use `A = copy(B)`.
- Julia values are not copied when passed to a function. If a function modifies an array, the changes will be visible in the caller.
- Julia does not automatically grow arrays in an assignment statement. Whereas in MATLAB `a(4) = 3.2` can create the array `a = [0 0 0 3.2]` and `a(5) = 7` can grow it into `a = [0 0 0 3.2 7]`, the corresponding Julia statement `a[5] = 7` throws an error if the length of `a` is less than 5 or if this statement is the first use of the identifier `a`. Julia has `push!` and `append!`, which grow `Vectors` much more efficiently than MATLAB's `a(end+1) = val`.
- The imaginary unit `sqrt(-1)` is represented in Julia as `im`, not `i` or `j` as in MATLAB.
- In Julia, literal numbers without a decimal point (such as 42) create integers instead of floating point numbers. As a result, some operations can throw a domain error if they expect a float; for example, `julia> a = -1; 2^a` throws a domain error, as the result is not an integer (see the FAQ entry on domain errors for details).
- In Julia, multiple values are returned and assigned as tuples, e.g. `(a, b) = (1, 2)` or `a, b = 1, 2`. MATLAB's `nargout`, which is often used in MATLAB to do optional work based on the number of returned values, does not exist in Julia. Instead, users can use optional and keyword arguments to achieve similar capabilities.
- Julia has true one-dimensional arrays. Column vectors are of size `N`, not `Nx1`. For example, `rand(N)` makes a 1-dimensional array.

- In Julia, `[x,y,z]` will always construct a 3-element array containing `x`, `y` and `z`.
  - To concatenate in the first ("vertical") dimension use either `vcat(x,y,z)` or separate with semicolons (`[x; y; z]`).
  - To concatenate in the second ("horizontal") dimension use either `hcat(x,y,z)` or separate with spaces (`[x y z]`).
  - To construct block matrices (concatenating in the first two dimensions), use either `hvcat` or combine spaces and semicolons (`[a b; c d]`).
- In Julia, `a:b` and `a:b:c` construct `AbstractRange` objects. To construct a full vector like in MATLAB, use `collect(a:b)`. Generally, there is no need to call `collect` though. An `AbstractRange` object will act like a normal array in most cases but is more efficient because it lazily computes its values. This pattern of creating specialized objects instead of full arrays is used frequently, and is also seen in functions such as `range`, or with iterators such as `enumerate`, and `zip`. The special objects can mostly be used as if they were normal arrays.
- Functions in Julia return values from their last expression or the `return` keyword instead of listing the names of variables to return in the function definition (see The return Keyword for details).
- A Julia script may contain any number of functions, and all definitions will be externally visible when the file is loaded. Function definitions can be loaded from files outside the current working directory.
- In Julia, reductions such as `sum`, `prod`, and `maximum` are performed over every element of an array when called with a single argument, as in `sum(A)`, even if `A` has more than one dimension.
- In Julia, parentheses must be used to call a function with zero arguments, like in `rand()`.
- Julia discourages the use of semicolons to end statements. The results of statements are not automatically printed (except at the interactive prompt), and lines of code do not need to end with semicolons. `println` or `@printf` can be used to print specific output.
- In Julia, if `A` and `B` are arrays, logical comparison operations like `A == B` do not return an array of booleans. Instead, use `A .== B`, and similarly for the other boolean operators like `<`, `>`.
- In Julia, when you want to apply a scalar-valued function elementwise to an array, use broadcasting syntax: `f.(A)` instead of `f(A)`. In some cases, both operations are defined but mean different things: in MATLAB `exp(A)` applies elementwise and `expm(A)` is the matrix exponential, but in Julia `exp.(A)` applies elementwise and `exp(A)` is the matrix exponential.
- In Julia, the operators `&`, `|`, and `⊻` (`xor`) perform the bitwise operations equivalent to `and`, `or`, and `xor` respectively in MATLAB, and have precedence similar to Python's bitwise operators (unlike C). To apply logical boolean operators over an array (like common uses of MATLAB's `&` and `|`), broadcast Julia's short-circuiting operators `.&&` and `.||`. For example, to test if the elements in an array `A` are equal to 1 or 2, you can use `A .== 1 .|| A .== 2`.
- In Julia, the elements of a collection can be passed as arguments to a function using the splat operator `...`, as in `xs=[1,2]; f(xs...)`.
- Julia's `svd` returns singular values as a vector instead of as a dense diagonal matrix.
- In Julia, `...` is not used to continue lines of code. Instead, incomplete expressions automatically continue onto the next line.

- In both Julia and MATLAB, the variable `ans` is set to the value of the last expression issued in an interactive session. In Julia, unlike MATLAB, `ans` is not set when Julia code is run in non-interactive mode.
- Julia's `structs` do not support dynamically adding fields at runtime, unlike MATLAB's `classes`. Instead, use a `Dict`. Dict in Julia isn't ordered.
- In Julia each module has its own global scope/namespace, whereas in MATLAB there is just one global scope.
- In MATLAB, an idiomatic way to remove unwanted values is to use logical indexing, like in the expression `x(x>3)` or in the statement `x(x>3) = []` to modify `x` in-place. In contrast, Julia provides the higher order functions `filter` and `filter!`, allowing users to write `filter(z->z>3, x)` and `filter!(z->z>3, x)` as alternatives to the corresponding transliterations `x[x.>3]` and `x = x[x.>3]`. Using `filter!` reduces the use of temporary arrays.
- Following on from the previous point, to replace values that meet specific criteria, for example a thresholding operation on all elements in a matrix, could be achieved in Matlab as follows `A(A < threshold) = 0`. The Julia equivalent would be `A[A .< threshold] .= 0`.
- The analogue of extracting (or "dereferencing") all elements of a cell array, e.g. in `vertcat(A{:})` in MATLAB, is written using the splat operator in Julia, e.g. as `vcat(A...)`.
- In Julia, the `adjoint` function performs conjugate transposition; in MATLAB, `adjoint` provides the "adjugate" or classical adjoint, which is the transpose of the matrix of cofactors.
- In Julia, a^b^c is evaluated a^(b^c) while in MATLAB it's (a^b)^c.

40.2 Noteworthy differences from R

One of Julia's goals is to provide an effective language for data analysis and statistical programming. For users coming to Julia from R, these are some noteworthy differences:

- Julia's single quotes enclose characters, not strings.
- Julia can create substrings by indexing into strings. In R, strings must be converted into character vectors before creating substrings.
- In Julia, like Python but unlike R, strings can be created with triple quotes `""" ... """`. This syntax is convenient for constructing strings that contain line breaks.
- In Julia, varargs are specified using the splat operator `...`, which always follows the name of a specific variable, unlike R, for which `...` can occur in isolation.
- In Julia, modulus is `mod(a, b)`, not `a %% b`. `%` in Julia is the remainder operator.
- Julia constructs vectors using brackets. Julia's `[1, 2, 3]` is the equivalent of R's `c(1, 2, 3)`.
- In Julia, not all data structures support logical indexing. Furthermore, logical indexing in Julia is supported only with vectors of length equal to the object being indexed. For example:
  - In R, `c(1, 2, 3, 4)[c(TRUE, FALSE)]` is equivalent to `c(1, 3)`.
  - In R, `c(1, 2, 3, 4)[c(TRUE, FALSE, TRUE, FALSE)]` is equivalent to `c(1, 3)`.

  - In Julia, `[1, 2, 3, 4][[true, false]]` throws a `BoundsError`.
  - In Julia, `[1, 2, 3, 4][[true, false, true, false]]` produces `[1, 3]`.

- Like many languages, Julia does not always allow operations on vectors of different lengths, unlike R where the vectors only need to share a common index range. For example, `c(1, 2, 3, 4) + c(1, 2)` is valid R but the equivalent `[1, 2, 3, 4] + [1, 2]` will throw an error in Julia.
- Julia allows an optional trailing comma when that comma does not change the meaning of code. This can cause confusion among R users when indexing into arrays. For example, `x[1,]` in R would return the first row of a matrix; in Julia, however, the comma is ignored, so `x[1,] == x[1]`, and will return the first element. To extract a row, be sure to use `:`, as in `x[1,:]`.
- Julia's `map` takes the function first, then its arguments, unlike `lapply(<structure>, function, ...)` in R. Similarly Julia's equivalent of `apply(X, MARGIN, FUN, ...)` in R is `mapslices` where the function is the first argument.
- Multivariate apply in R, e.g. `mapply(choose, 11:13, 1:3)`, can be written as `broadcast(binomial, 11:13, 1:3)` in Julia. Equivalently Julia offers a shorter dot syntax for vectorizing functions `binomial.(11:13, 1:3)`.
- Julia uses `end` to denote the end of conditional blocks, like `if`, loop blocks, like `while`/ `for`, and functions. In lieu of the one-line `if ( cond ) statement`, Julia allows statements of the form `if cond; statement; end`, `cond && statement` and `!cond || statement`. Assignment statements in the latter two syntaxes must be explicitly wrapped in parentheses, e.g. `cond && (x = value)`.
- In Julia, `<-`, `<<-` and `->` are not assignment operators.
- Julia's `->` creates an anonymous function.
- Julia's `*` operator can perform matrix multiplication, unlike in R. If `A` and `B` are matrices, then `A * B` denotes a matrix multiplication in Julia, equivalent to R's `A %*% B`. In R, this same notation would perform an element-wise (Hadamard) product. To get the element-wise multiplication operation, you need to write `A .* B` in Julia.
- Julia performs matrix transposition using the `transpose` function and conjugated transposition using the `'` operator or the `adjoint` function. Julia's `transpose(A)` is therefore equivalent to R's `t(A)`. Additionally a non-recursive transpose in Julia is provided by the `permutedims` function.
- Julia does not require parentheses when writing `if` statements or `for`/`while` loops: use `for i in [1, 2, 3]` instead of `for (i in c(1, 2, 3))` and `if i == 1` instead of `if (i == 1)`.
- Julia does not treat the numbers `0` and `1` as Booleans. You cannot write `if (1)` in Julia, because `if` statements accept only booleans. Instead, you can write `if true`, `if Bool(1)`, or `if 1==1`.
- Julia does not provide `nrow` and `ncol`. Instead, use `size(M, 1)` for `nrow(M)` and `size(M, 2)` for `ncol(M)`.
- Julia is careful to distinguish scalars, vectors and matrices. In R, `1` and `c(1)` are the same. In Julia, they cannot be used interchangeably.
- Julia's `diag` and `diagm` are not like R's.
- Julia cannot assign to the results of function calls on the left hand side of an assignment operation: you cannot write `diag(M) = fill(1, n)`.

- Julia discourages populating the main namespace with functions. Most statistical functionality for Julia is found in packages under the JuliaStats organization. For example:
  - Functions pertaining to probability distributions are provided by the Distributions package.
  - The DataFrames package provides data frames.
  - Generalized linear models are provided by the GLM package.
- Julia provides tuples and real hash tables, but not R-style lists. When returning multiple items, you should typically use a tuple or a named tuple: instead of `list(a = 1, b = 2)`, use `(1, 2)` or `(a=1, b=2)`.
- Julia encourages users to write their own types, which are easier to use than S3 or S4 objects in R. Julia's multiple dispatch system means that `table(x::TypeA)` and `table(x::TypeB)` act like R's `table.TypeA(x)` and `table.TypeB(x)`.
- In Julia, values are not copied when assigned or passed to a function. If a function modifies an array, the changes will be visible in the caller. This is very different from R and allows new functions to operate on large data structures much more efficiently.
- In Julia, vectors and matrices are concatenated using `hcat`, `vcat` and `hvcat`, not `c`, `rbind` and `cbind` like in R.
- In Julia, a range like `a:b` is not shorthand for a vector like in R, but is a specialized `AbstractRange` object that is used for iteration. To convert a range into a vector, use `collect(a:b)`.
- The `:` operator has a different precedence in R and Julia. In particular, in Julia arithmetic operators have higher precedence than the `:` operator, whereas the reverse is true in R. For example, `1:n-1` in Julia is equivalent to `1:(n-1)` in R.
- Julia's `max` and `min` are the equivalent of `pmax` and `pmin` respectively in R, but both arguments need to have the same dimensions. While `maximum` and `minimum` replace `max` and `min` in R, there are important differences.
- Julia's `sum`, `prod`, `maximum`, and `minimum` are different from their counterparts in R. They all accept an optional keyword argument `dims`, which indicates the dimensions, over which the operation is carried out. For instance, let `A = [1 2; 3 4]` in Julia and `B <- rbind(c(1,2),c(3,4))` be the same matrix in R. Then `sum(A)` gives the same result as `sum(B)`, but `sum(A, dims=1)` is a row vector containing the sum over each column and `sum(A, dims=2)` is a column vector containing the sum over each row. This contrasts to the behavior of R, where separate `colSums(B)` and `rowSums(B)` functions provide these functionalities. If the `dims` keyword argument is a vector, then it specifies all the dimensions over which the sum is performed, while retaining the dimensions of the summed array, e.g. `sum(A, dims=(1,2)) == hcat(10)`. It should be noted that there is no error checking regarding the second argument.
- Julia has several functions that can mutate their arguments. For example, it has both `sort` and `sort!`.
- In R, performance requires vectorization. In Julia, almost the opposite is true: the best performing code is often achieved by using devectorized loops.
- Julia is eagerly evaluated and does not support R-style lazy evaluation. For most users, this means that there are very few unquoted expressions or column names.

- Julia does not support the NULL type. The closest equivalent is `nothing`, but it behaves like a scalar value rather than like a list. Use `x === nothing` instead of `is.null(x)`.
- In Julia, missing values are represented by the `missing` object rather than by NA. Use `ismissing(x)` (or `ismissing.(x)` for element-wise operation on vectors) instead of `is.na(x)`. The `skipmissing` function is generally used instead of `na.rm=TRUE` (though in some particular cases functions take a `skipmissing` argument).
- Julia lacks the equivalent of R's `assign` or `get`.
- In Julia, `return` does not require parentheses.
- In R, an idiomatic way to remove unwanted values is to use logical indexing, like in the expression `x[x>3]` or in the statement `x = x[x>3]` to modify x in-place. In contrast, Julia provides the higher order functions `filter` and `filter!`, allowing users to write `filter(z->z>3, x)` and `filter!(z->z>3, x)` as alternatives to the corresponding transliterations `x[x.>3]` and `x = x[x.>3]`. Using `filter!` reduces the use of temporary arrays.

40.3 Noteworthy differences from Python

- Julia's `for`, `if`, `while`, etc. blocks are terminated by the end keyword. Indentation level is not significant as it is in Python. Unlike Python, Julia has no `pass` keyword.
- Strings are denoted by double quotation marks (`"text"`) in Julia (with three double quotation marks for multi-line strings), whereas in Python they can be denoted either by single (`'text'`) or double quotation marks (`"text"`). Single quotation marks are used for characters in Julia (`'c'`).
- String concatenation is done with * in Julia, not + like in Python. Analogously, string repetition is done with ^, not *. Implicit string concatenation of string literals like in Python (e.g. `'ab' 'cd' == 'abcd'`) is not done in Julia.
- Python Lists—flexible but slow—correspond to the Julia `Vector{Any}` type or more generally `Vector{T}` where T is some non-concrete element type. "Fast" arrays like NumPy arrays that store elements in-place (i.e., `dtype` is `np.float64`, `[('f1', np.uint64), ('f2', np.int32)]`, etc.) can be represented by `Array{T}` where T is a concrete, immutable element type. This includes built-in types like `Float64`, `Int32`, `Int64` but also more complex types like `Tuple{UInt64,Float64}` and many user-defined types as well.
- In Julia, indexing of arrays, strings, etc. is 1-based not 0-based.
- Julia's slice indexing includes the last element, unlike in Python. `a[2:3]` in Julia is `a[1:3]` in Python.
- Unlike Python, Julia allows AbstractArrays with arbitrary indexes. Python's special interpretation of negative indexing, `a[-1]` and `a[-2]`, should be written `a[end]` and `a[end-1]` in Julia.
- Julia requires end for indexing until the last element. `x[2:end]` in Julia is equivalent to `x[1:]` in Python.
- In Julia, `:` before any object creates a `Symbol` or *quotes* an expression; so, `x[:5]` is the same as `x[5]`. If you want to get the first n elements of an array, then use range indexing.
- Julia's range indexing has the format of `x[start:step:stop]`, whereas Python's format is `x[start:(stop+1):step]`. Hence, `x[0:10:2]` in Python is equivalent to `x[1:2:10]` in Julia. Similarly, `x[::-1]` in Python, which refers to the reversed array, is equivalent to `x[end:-1:1]` in Julia.

- In Julia, ranges can be constructed independently as `start:step:stop`, the same syntax it uses in array-indexing. The `range` function is also supported.
- In Julia, indexing a matrix with arrays like `X[[1,2], [1,3]]` refers to a sub-matrix that contains the intersections of the first and second rows with the first and third columns. In Python, `X[[1,2], [1,3]]` refers to a vector that contains the values of cell `[1,1]` and `[2,3]` in the matrix. `X[[1,2], [1,3]]` in Julia is equivalent with `X[np.ix_([0,1],[0,2])]` in Python. `X[[0,1], [0,2]]` in Python is equivalent with `X[[CartesianIndex(1,1), CartesianIndex(2,3)]]` in Julia.
- Julia has no line continuation syntax: if, at the end of a line, the input so far is a complete expression, it is considered done; otherwise the input continues. One way to force an expression to continue is to wrap it in parentheses.
- Julia arrays are column-major (Fortran-ordered) whereas NumPy arrays are row-major (C-ordered) by default. To get optimal performance when looping over arrays, the order of the loops should be reversed in Julia relative to NumPy (see relevant section of Performance Tips).
- Julia's updating operators (e.g. `+=`, `-=`, ...) are *not in-place* whereas NumPy's are. This means `A = [1, 1]; B = A; B += [3, 3]` doesn't change values in `A`, it rather rebinds the name `B` to the result of the right-hand side `B = B + 3`, which is a new array. For in-place operation, use `B .+= 3` (see also dot operators), explicit loops, or `InplaceOps.jl`.
- Julia evaluates default values of function arguments every time the method is invoked, unlike in Python where the default values are evaluated only once when the function is defined. For example, the function `f(x=rand()) = x` returns a new random number every time it is invoked without argument. On the other hand, the function `g(x=[1,2]) = push!(x,3)` returns `[1,2,3]` every time it is called as `g()`.
- In Julia, keyword arguments must be passed using keywords, unlike Python in which it is usually possible to pass them positionally. Attempting to pass a keyword argument positionally alters the method signature leading to a `MethodError` or calling of the wrong method.
- In Julia `%` is the remainder operator, whereas in Python it is the modulus.
- In Julia, the commonly used `Int` type corresponds to the machine integer type (`Int32` or `Int64`), unlike in Python, where `int` is an arbitrary length integer. This means in Julia the `Int` type will overflow, such that `2^64 == 0`. If you need larger values use another appropriate type, such as `Int128`, `BigInt` or a floating point type like `Float64`.
- The imaginary unit `sqrt(-1)` is represented in Julia as `im`, not `j` as in Python.
- In Julia, the exponentiation operator is `^`, not `**` as in Python.
- Julia uses `nothing` of type `Nothing` to represent a null value, whereas Python uses `None` of type `NoneType`.
- In Julia, the standard operators over a matrix type are matrix operations, whereas, in Python, the standard operators are element-wise operations. When both `A` and `B` are matrices, `A * B` in Julia performs matrix multiplication, not element-wise multiplication as in Python. `A * B` in Julia is equivalent with `A @ B` in Python, whereas `A * B` in Python is equivalent with `A .* B` in Julia.
- In Julia, when you want to apply a scalar-valued function elementwise to an array, use broadcasting syntax: `f.(A)` instead of `f(A)`. In some cases, both operations are defined but mean different things: `numpy.exp(A)` applies elementwise and `scipy.linalg.expm(A)` is the matrix exponential, but in Julia `exp.(A)` applies elementwise and `exp(A)` is the matrix exponential.

- The adjoint operator `'` in Julia returns an adjoint of a vector (a lazy representation of row vector), whereas the transpose operator `.T` over a vector in Python returns the original vector (non-op).
- In Julia, a function may contain multiple concrete implementations (called *methods*), which are selected via multiple dispatch based on the types of all arguments to the call, as compared to functions in Python, which have a single implementation and no polymorphism (as opposed to Python method calls which use a different syntax and allows dispatch on the receiver of the method).
- There are no classes in Julia. Instead there are structures (mutable or immutable), containing data but no methods.
- Calling a method of a class instance in Python (`x = MyClass(*args); x.f(y)`) corresponds to a function call in Julia, e.g. `x = MyType(args...); f(x, y)`. In general, multiple dispatch is more flexible and powerful than the Python class system.
- Julia structures may have exactly one abstract supertype, whereas Python classes can inherit from one or more (abstract or concrete) superclasses.
- The logical Julia program structure (Packages and Modules) is independent of the file structure, whereas the Python code structure is defined by directories (Packages) and files (Modules).
- In Julia, it is idiomatic to split the text of large modules into multiple files, without introducing a new module per file. The code is reassembled inside a single module in a main file via `include`. While the Python equivalent (exec) is not typical for this use (it will silently clobber prior definitions), Julia programs are defined as a unit at the `module` level with `using` or `import`, which will only get executed once when first needed–like `include` in Python. Within those modules, the individual files that make up that module are loaded with `include` by listing them once in the intended order.
- The ternary operator `x > 0 ? 1 : -1` in Julia corresponds to a conditional expression in Python `1 if x > 0 else -1`.
- In Julia the `@` symbol refers to a macro, whereas in Python it refers to a decorator.
- Exception handling in Julia is done using `try — catch — finally`, instead of `try — except — finally`. In contrast to Python, it is not recommended to use exception handling as part of the normal workflow in Julia (compared with Python, Julia is faster at ordinary control flow but slower at exception-catching).
- In Julia loops are fast, there is no need to write "vectorized" code for performance reasons.
- Be careful with non-constant global variables in Julia, especially in tight loops. Since you can write close-to-metal code in Julia (unlike Python), the effect of globals can be drastic (see Performance Tips).
- In Julia, rounding and truncation are explicit. Python's `int(3.7)` should be `floor(Int, 3.7)` or `Int(floor(3.7))` and is distinguished from `round(Int, 3.7)`. `floor(x)` and `round(x)` on their own return an integer value of the same type as `x` rather than always returning `Int`.
- In Julia, parsing is explicit. Python's `float("3.7")` would be `parse(Float64, "3.7")` in Julia.
- In Python, the majority of values can be used in logical contexts (e.g. `if "a":` means the following block is executed, and `if "":` means it is not). In Julia, you need explicit conversion to `Bool` (e.g. `if "a"` throws an exception). If you want to test for a non-empty string in Julia, you would explicitly write `if !isempty("")`. Perhaps surprisingly, in Python `if "False"` and `bool("False")` both evaluate to `True` (because `"False"` is a non-empty string); in Julia, `parse(Bool, "false")` returns `false`.

- In Julia, a new local scope is introduced by most code blocks, including loops and `try — catch — finally`. Note that comprehensions (list, generator, etc.) introduce a new local scope both in Python and Julia, whereas `if` blocks do not introduce a new local scope in both languages.

40.4 Noteworthy differences from C/C++

- Julia arrays are indexed with square brackets, and can have more than one dimension `A[i,j]`. This syntax is not just syntactic sugar for a reference to a pointer or address as in C/C++. See the manual entry about array construction.
- In Julia, indexing of arrays, strings, etc. is 1-based not 0-based.
- Julia arrays are not copied when assigned to another variable. After `A = B`, changing elements of `B` will modify `A` as well. Updating operators like `+=` do not operate in-place, they are equivalent to `A = A + B` which rebinds the left-hand side to the result of the right-hand side expression.
- Julia arrays are column major (Fortran ordered) whereas C/C++ arrays are row major ordered by default. To get optimal performance when looping over arrays, the order of the loops should be reversed in Julia relative to C/C++ (see relevant section of Performance Tips).
- Julia values are not copied when assigned or passed to a function. If a function modifies an array, the changes will be visible in the caller.
- In Julia, whitespace is significant, unlike C/C++, so care must be taken when adding/removing whitespace from a Julia program.
- In Julia, literal numbers without a decimal point (such as 42) create signed integers, of type `Int`, but literals too large to fit in the machine word size will automatically be promoted to a larger size type, such as `Int64` (if `Int` is `Int32`), `Int128`, or the arbitrarily large `BigInt` type. There are no numeric literal suffixes, such as `L`, `LL`, `U`, `UL`, `ULL` to indicate unsigned and/or signed vs. unsigned. Decimal literals are always signed, and hexadecimal literals (which start with `0x` like C/C++), are unsigned, unless when they encode more than 128 bits, in which case they are of type `BigInt`. Hexadecimal literals also, unlike C/C++/Java and unlike decimal literals in Julia, have a type based on the *length* of the literal, including leading 0s. For example, `0x0` and `0x00` have type `UInt8`, `0x000` and `0x0000` have type `UInt16`, then literals with 5 to 8 hex digits have type `UInt32`, 9 to 16 hex digits type `UInt64`, 17 to 32 hex digits type `UInt128`, and more that 32 hex digits type `BigInt`. This needs to be taken into account when defining hexadecimal masks, for example `~0xf == 0xf0` is very different from `~0x000f == 0xfff0`. 64 bit `Float64` and 32 bit `Float32` bit literals are expressed as `1.0` and `1.0f0` respectively. Floating point literals are rounded (and not promoted to the `BigFloat` type) if they can not be exactly represented. Floating point literals are closer in behavior to C/C++. Octal (prefixed with `0o`) and binary (prefixed with `0b`) literals are also treated as unsigned (or `BigInt` for more than 128 bits).
- In Julia, the division operator `/` returns a floating point number when both operands are of integer type. To perform integer division, use `div` or `÷`.
- Indexing an `Array` with floating point types is generally an error in Julia. The Julia equivalent of the C expression `a[i / 2]` is `a[i ÷ 2 + 1]`, where `i` is of integer type.
- String literals can be delimited with either `"` or `"""`, `"""` delimited literals can contain `"` characters without quoting it like `"\""`. String literals can have values of other variables or expressions interpolated into them, indicated by `$variablename` or `$(expression)`, which evaluates the variable name or the expression in the context of the function.

- `//` indicates a `Rational` number, and not a single-line comment (which is # in Julia)
- #= indicates the start of a multiline comment, and =# ends it.
- Functions in Julia return values from their last expression(s) or the `return` keyword. Multiple values can be returned from functions and assigned as tuples, e.g. `(a, b) = myfunction()` or `a, b = myfunction()`, instead of having to pass pointers to values as one would have to do in C/C++ (i.e. `a = myfunction(&b)`.
- Julia does not require the use of semicolons to end statements. The results of expressions are not automatically printed (except at the interactive prompt, i.e. the REPL), and lines of code do not need to end with semicolons. `println` or `@printf` can be used to print specific output. In the REPL, `;` can be used to suppress output. `;` also has a different meaning within `[ ]`, something to watch out for. `;` can be used to separate expressions on a single line, but are not strictly necessary in many cases, and are more an aid to readability.
- In Julia, the operator `⊻` (`xor`) performs the bitwise XOR operation, i.e. `^` in C/C++. Also, the bitwise operators do not have the same precedence as C/C++, so parenthesis may be required.
- Julia's `^` is exponentiation (pow), not bitwise XOR as in C/C++ (use `⊻`, or `xor`, in Julia)
- Julia has two right-shift operators, >> and >>>. >> performs an arithmetic shift, >>> always performs a logical shift, unlike C/C++, where the meaning of >> depends on the type of the value being shifted.
- Julia's `->` creates an anonymous function, it does not access a member via a pointer.
- Julia does not require parentheses when writing `if` statements or `for`/`while` loops: use `for i in [1, 2, 3]` instead of `for (int i=1; i <= 3; i++)` and `if i == 1` instead of `if (i == 1)`.
- Julia does not treat the numbers `0` and `1` as Booleans. You cannot write `if (1)` in Julia, because `if` statements accept only booleans. Instead, you can write `if true`, `if Bool(1)`, or `if 1==1`.
- Julia uses `end` to denote the end of conditional blocks, like `if`, loop blocks, like `while`/ `for`, and functions. In lieu of the one-line `if ( cond ) statement`, Julia allows statements of the form `if cond; statement; end`, `cond && statement` and `!cond || statement`. Assignment statements in the latter two syntaxes must be explicitly wrapped in parentheses, e.g. `cond && (x = value)`, because of the operator precedence.
- Julia has no line continuation syntax: if, at the end of a line, the input so far is a complete expression, it is considered done; otherwise the input continues. One way to force an expression to continue is to wrap it in parentheses.
- Julia macros operate on parsed expressions, rather than the text of the program, which allows them to perform sophisticated transformations of Julia code. Macro names start with the `@` character, and have both a function-like syntax, `@mymacro(arg1, arg2, arg3)`, and a statement-like syntax, `@mymacro arg1 arg2 arg3`. The forms are interchangeable; the function-like form is particularly useful if the macro appears within another expression, and is often clearest. The statement-like form is often used to annotate blocks, as in the distributed `for` construct: `@distributed for i in 1:n; #= body =#; end`. Where the end of the macro construct may be unclear, use the function-like form.
- Julia has an enumeration type, expressed using the macro `@enum(name, value1, value2, ...)` For example: `@enum(Fruit, banana=1, apple, pear)`

- By convention, functions that modify their arguments have a `!` at the end of the name, for example `push!`.
- In C++, by default, you have static dispatch, i.e. you need to annotate a function as virtual, in order to have dynamic dispatch. On the other hand, in Julia every method is "virtual" (although it's more general than that since methods are dispatched on every argument type, not only `this`, using the most-specific-declaration rule).

Julia ⇔ C/C++: Namespaces

- C/C++ `namespaces` correspond roughly to Julia `modules`.
- There are no private globals or fields in Julia. Everything is publicly accessible through fully qualified paths (or relative paths, if desired).
- `using MyNamespace::myfun` (C++) corresponds roughly to `import MyModule: myfun` (Julia).
- `using namespace MyNamespace` (C++) corresponds roughly to `using MyModule` (Julia)
  - In Julia, only `exported` symbols are made available to the calling module.
  - In C++, only elements found in the included (public) header files are made available.
- Caveat: `import`/`using` keywords (Julia) also *load* modules (see below).
- Caveat: `import`/`using` (Julia) works only at the global scope level (`modules`)
  - In C++, `using namespace X` works within arbitrary scopes (ex: function scope).

Julia ⇔ C/C++: Module loading

- When you think of a C/C++ "**library**", you are likely looking for a Julia "**package**".
  - Caveat: C/C++ libraries often house multiple "software modules" whereas Julia "packages" typically house one.
  - Reminder: Julia `modules` are global scopes (not necessarily "software modules").
- **Instead of build/`make` scripts**, Julia uses "Project Environments" (sometimes called either "Project" or "Environment").
  - Build scripts are only needed for more complex applications (like those needing to compile or download C/C++ executables).
  - To develop application or project in Julia, you can initialize its root directory as a "Project Environment", and house application-specific code/packages there. This provides good control over project dependencies, and future reproducibility.
  - Available packages are added to a "Project Environment" with the `Pkg.add()` function or Pkg REPL mode. (This does not **load** said package, however).
  - The list of available packages (direct dependencies) for a "Project Environment" are saved in its `Project.toml` file.
  - The *full* dependency information for a "Project Environment" is auto-generated & saved in its `Manifest.toml` file by `Pkg.resolve()`.

- Packages ("software modules") available to the "Project Environment" are loaded with `import` or `using`.
  - In C/C++, you `#include <moduleheader>` to get object/function declarations, and link in libraries when you build the executable.
  - In Julia, calling using/import again just brings the existing module into scope, but does not load it again (similar to adding the non-standard `#pragma once` to C/C++).
- **Directory-based package repositories** (Julia) can be made available by adding repository paths to the `Base.LOAD_PATH` array.
  - Packages from directory-based repositories do not require the `Pkg.add()` tool prior to being loaded with `import` or `using`. They are simply available to the project.
  - Directory-based package repositories are the **quickest solution** to developing local libraries of "software modules".

Julia ⇔ C/C++: Assembling modules

- In C/C++, `.c`/`.cpp` files are compiled & added to a library with build/make scripts.
  - In Julia, `import [PkgName]`/`using [PkgName]` statements load `[PkgName].jl` located in a package's `[PkgName]/src/` subdirectory.
  - In turn, `[PkgName].jl` typically loads associated source files with calls to `include "[someotherfile].jl"`.
- `include "./path/to/somefile.jl"` (Julia) is very similar to `#include "./path/to/somefile.jl"` (C/C++).
  - However `include "..."` (Julia) is not used to include header files (not required).
  - **Do not use** `include "..."` (Julia) to load code from other "software modules" (use `import`/`using` instead).
  - `include "path/to/some/module.jl"` (Julia) would instantiate multiple versions of the same code in different modules (creating *distinct* types (etc.) with the *same* names).
  - `include "somefile.jl"` is typically used to assemble multiple files *within the same Julia package* ("software module"). It is therefore relatively straightforward to ensure file are `included` only once (No `#ifdef` confusion).

Julia ⇔ C/C++: Module interface

- C++ exposes interfaces using "public" `.h`/`.hpp` files whereas Julia `modules` mark specific symbols that are intended for their users as `public`or `exported`.
  - Often, Julia `modules` simply add functionality by generating new "methods" to existing functions (ex: `Base.push!`).
  - Developers of Julia packages therefore cannot rely on header files for interface documentation.
  - Interfaces for Julia packages are typically described using docstrings, README.md, static web pages, ...
- Some developers choose not to `export` all symbols required to use their package/module, but should still mark unexported user facing symbols as `public`.
  - Users might be expected to access these components by qualifying functions/structs/... with the package/module name (ex: `MyModule.run_this_task(...)`).

Julia ⇔ C/C++: Quick reference

Software Concept	Julia	C/C++
unnamed scope	`begin ... end`	`{ ... }`
function scope	`function x() ... end`	`int x() { ... }`
global scope	`module MyMod ... end`	`namespace MyNS { ... }`
software module	A Julia "package"	`.h/.hpp` files +compiled `somelib.a`

assembling
software modules
	`SomePkg.jl`: ...
	`import("subfile1.jl")`
	`import("subfile2.jl")`
	...
		`$(AR) *.o` ⇒ `somelib.a`

import software module	`import SomePkg`	`#include <somelib>` +link in `somelib.a`
module library	`LOAD_PATH[]`, *Git repository, **custom package registry	more `.h/.hpp` files +bigger compiled `somebiglib.a`

* The Julia package manager supports registering multiple packages from a single Git repository.
 * This allows users to house a library of related packages in a single repository.
 ** Julia registries are primarily designed to provide versioning \& distribution of packages.
 ** Custom package registries can be used to create a type of module library.

40.5 Noteworthy differences from Common Lisp

- Julia uses 1-based indexing for arrays by default, and it can also handle arbitrary index offsets.
- Functions and variables share the same namespace ("Lisp-1").
- There is a `Pair` type, but it is not meant to be used as a `COMMON-LISP:CONS`. Various iterable collections can be used interchangeably in most parts of the language (eg splatting, tuples, etc). `Tuples` are the closest to Common Lisp lists for *short* collections of heterogeneous elements. Use `NamedTuples` in place of alists. For larger collections of homogeneous types, `Arrays` and `Dicts` should be used.
- The typical Julia workflow for prototyping also uses continuous manipulation of the image, implemented with the Revise.jl package.
- For performance, Julia prefers that operations have type stability. Where Common Lisp abstracts away from the underlying machine operations, Julia cleaves closer to them. For example:
  - Integer division using `/` always returns a floating-point result, even if the computation is exact.
    - `//` always returns a rational result
    - `÷` always returns a (truncated) integer result
  - Bignums are supported, but conversion is not automatic; ordinary integers overflow.
  - Complex numbers are supported, but to get complex results, you need complex inputs.
  - There are multiple Complex and Rational types, with different component types.

- Modules (namespaces) can be hierarchical. `import` and `using` have a dual role: they load the code and make it available in the namespace. `import` for only the module name is possible (roughly equivalent to `ASDF:LOAD-OP`). Slot names don't need to be exported separately. Global variables can't be assigned to from outside the module (except with `eval(mod, :(var = val))` as an escape hatch).
- Macros start with `@`, and are not as seamlessly integrated into the language as Common Lisp; consequently, macro usage is not as widespread as in the latter. A form of hygiene for macros is supported by the language. Because of the different surface syntax, there is no equivalent to `COMMON-LISP:&BODY`.
- *All* functions are generic and use multiple dispatch. Argument lists don't have to follow the same template, which leads to a powerful idiom (see `do`). Optional and keyword arguments are handled differently. Method ambiguities are not resolved like in the Common Lisp Object System, necessitating the definition of a more specific method for the intersection.
- Symbols do not belong to any package, and do not contain any values *per se*. `M.var` evaluates the symbol `var` in the module `M`.
- A functional programming style is fully supported by the language, including closures, but isn't always the idiomatic solution for Julia. Some workarounds may be necessary for performance when modifying captured variables.

Chapter 41

Unicode Input

The following table lists Unicode characters that can be entered via tab completion of LaTeX-like abbreviations in the Julia REPL (and in various other editing environments). You can also get information on how to type a symbol by entering it in the REPL help, i.e. by typing ? and then entering the symbol in the REPL (e.g., by copy-paste from somewhere you saw the symbol).

Warning

This table may appear to contain missing characters in the second column, or even show characters that are inconsistent with the characters as they are rendered in the Julia REPL. In these cases, users are strongly advised to check their choice of fonts in their browser and REPL environment, as there are known issues with glyphs in many fonts.

Chapter 42

Command-line Interface

42.1 Using arguments inside scripts

When running a script using `julia`, you can pass additional arguments to your script:

```
$ julia script.jl arg1 arg2...
```

These additional command-line arguments are passed in the global constant `ARGS`. The name of the script itself is passed in as the global `PROGRAM_FILE`. Note that `ARGS` is also set when a Julia expression is given using the `-e` option on the command line (see the `julia` help output below) but `PROGRAM_FILE` will be empty. For example, to just print the arguments given to a script, you could do this:

```
$ julia -e 'println(PROGRAM_FILE); for x in ARGS; println(x); end' foo bar

foo
bar
```

Or you could put that code into a script and run it:

```
$ echo 'println(PROGRAM_FILE); for x in ARGS; println(x); end' > script.jl
$ julia script.jl foo bar
script.jl
foo
bar
```

The `--` delimiter can be used to separate command-line arguments intended for the script file from arguments intended for Julia:

```
$ julia --color=yes -O -- script.jl arg1 arg2..
```

See also Scripting for more information on writing Julia scripts.

42.2 The `Main.main` entry point

As of Julia 1.11, `Base` exports the macro `@main`. This macro expands to the symbol `main`, but at the conclusion of executing a script or expression, `julia` will attempt to execute `Main.main(Base.ARGS)` if such a function `Main.main` has been defined and this behavior was opted into by using the `@main` macro.

To see this feature in action, consider the following definition:

```
(@main)(args) = println("Hello $(args[1])!")
```

Executing the above script with `julia script.jl "Buddy"` will automatically run `(@main)` and print "Hello Buddy!", despite there being no explicit call to `(@main)`.

The return value of the `(@main)` function must either be `nothing`, resulting in exit code `0`, or convertible to a `Cint` which will be the exit code:

```
$ julia -e "(@main)(args) = nothing"; echo $?0
0
$ julia -e "(@main)(args) = 1"; echo $?
1
```

Typically exit codes are in the range `0:255`, although the interpretation of the return value might be OS dependent.

This feature is intended to aid in the unification of compiled and interactive workflows. In compiled workflows, loading the code that defines the `main` function may be spatially and temporally separated from the invocation. However, for interactive workflows, the behavior is equivalent to explicitly calling `exit(main(ARGS))` at the end of the evaluated script or expression.

Julia 1.11

The special entry point `Main.main` was added in Julia 1.11. For compatibility with prior julia versions, add an explicit `@isdefined(var"@main") ? (@main) : exit(main(ARGS))` at the end of your scripts.

Only the `main` binding in the `Main` module has this behavior and only if the macro `@main` was used within the defining module.

For example, using `hello` instead of `main` will not result in the `hello` function executing:

```
$ julia -e 'hello(args) = println("Hello World!")'
$
```

and neither will a plain definition of `main`:

```
$ julia -e 'main(args) = println("Hello World!")'
$
```

However, the opt-in need not occur at definition time:

```
$ julia -e 'main(args) = println("Hello World!"); @main'
Hello World!
$
```

The `main` binding may be imported from a package. A *hello world* package defined as

```
module Hello

export main
(@main)(args) = println("Hello from the package!")

end
```

may be used as:

```
$ julia -e 'using Hello'
Hello from the package!
$ julia -e 'import Hello' # N.B.: Execution depends on the binding not whether the package is
↪  loaded
$
```

However, note that the current best practice recommendation is to not mix application and reusable library code in the same package. Helper applications may be distributed as separate packages or as scripts with separate `main` entry points in a package's `bin` folder.

42.3 Parallel mode

Julia can be started in parallel mode with either the `-p` or the `--machine-file` options. `-p n` will launch an additional `n` worker processes, while `--machine-file file` will launch a worker for each line in file `file`. The machines defined in `file` must be accessible via a password-less `ssh` login, with Julia installed at the same location as the current host. Each machine definition takes the form `[count*][user@]host[:port] [bind_addr[:port]]`. `user` defaults to current user, `port` to the standard ssh port. `count` is the number of workers to spawn on the node, and defaults to 1. The optional `bind-to bind_addr[:port]` specifies the IP address and port that other workers should use to connect to this worker.

42.4 Startup file

If you have code that you want executed whenever Julia is run, you can put it in `~/.julia/config/startup.jl`:

```
$ echo 'println("Greetings! □□! □□□□□?")' > ~/.julia/config/startup.jl
$ julia
Greetings! □□! □□□□□?

...
```

Note that although you should have a `~/.julia` directory once you've run Julia for the first time, you may need to create the `~/.julia/config` folder and the `~/.julia/config/startup.jl` file if you use it.

To have startup code run only in The Julia REPL (and not when `julia` is *e.g.* run on a script), use `atreplinit` in `startup.jl`:

```
atreplinit() do repl
    # ...
end
```

If JULIA_DEPOT_PATH is set, the startup file should be located there: `$JULIA_DEPOT_PATH/config/startup.jl`.

42.5 Command-line switches for Julia

There are various ways to run Julia code and provide options, similar to those available for the `perl` and `ruby` programs:

```
julia [switches] -- [programfile] [args...]
```

The following is a complete list of command-line switches available when launching julia (a '*' marks the default value, if applicable; settings marked '($)' may trigger package precompilation):

Options that have the form `--option={...}` can be specified either as `--option=value` or as `--option value`. For example, `julia --banner=no` is equivalent to `julia --banner no`. This is especially relevant for options that take a filename for output, because forgetting to specifying the argument for (say) `--trace-compile` will cause the option following it to be interpreted as the filename, possibly unintentionally overwriting it.

Note that options of the form `--option[=...]` can **not** be specified as `--option value`, but only as `--option=value` (or simply `--option`, when no argument is provided).

Julia 1.1

In Julia 1.0, the default `--project=@.` option did not search up from the root directory of a Git repository for the `Project.toml` file. From Julia 1.1 forward, it does.

Switch	Description
`-v, --version`	Display version information
`-h, --help`	Print command-line options (this message)
`--help-hidden`	Print uncommon options not shown by `-h`
`--project[={<dir>|@temp[@.]}]`	Set `<dir>` as the active project/environment. Or, create a temporary environment with `@temp`. The default `@.` option will search through parent directories until a `Project.toml` or `JuliaProject.toml` file is found.
`-J, --sysimage <file>`	Start up with the given system image file
`-H, --home <dir>`	Set location of `julia` executable
`--startup-file={yes*|no}`	Load `JULIA_DEPOT_PATH/config/startup.jl`; if JULIA_DEPOT_PATH environment variable is unset, load `~/.julia/config/startup.jl`
`--handle-signals={yes*|no}`	Enable or disable Julia's default signal handlers
`--sysimage-native-code={yes*|no}`	Use native code from system image if available
`--compiled-modules={yes*|no|existing|strict}`	Enable or disable incremental precompilation of modules. The `existing` option allows use of existing compiled modules that were previously precompiled, but disallows creation of new precompile files. The `strict` option is similar, but will error if no precompile file is found.
`--pkgimages={yes*|no|existing}`	Enable or disable usage of native code caching in the form of pkgimages. The `existing` option allows use of existing pkgimages but disallows creation of new ones
`-e, --eval <expr>`	Evaluate `<expr>`
`-E, --print <expr>`	Evaluate `<expr>` and display the result
`-m, --module <Package> [args]`	Run entry point of `Package` (`@main` function) with `args`
`-L, --load <file>`	Load `<file>` immediately on all processors
`-t, --threads {auto|N[,auto|M]}`	Enable N[+M] threads; N threads are assigned to the `default` threadpool, and if M is specified, M threads are assigned to the `interactive` threadpool; `auto` tries to infer a useful default number of threads to use but the exact behavior might change in the future. Currently sets N to the number of CPUs assigned to this Julia process based on the OS-specific affinity assignment interface if supported (Linux and Windows) or to the number of CPU threads if not supported (MacOS) or if process affinity is not configured, and sets M to 1.
`--gcthreads=N[,M]`	Use N threads for the mark phase of GC and M (0 or 1) threads for the concurrent sweeping phase of GC. N is set to the number of compute threads and M is set to 0 if unspecified. See Memory Management and Garbage Collection for more details.
`-p, --procs {N|auto}`	Integer value N launches N additional local worker processes; `auto` launches as many workers as the number of local CPU threads (logical cores)
`--machine-file <file>`	Run processes on hosts listed in `<file>`
`-i, --interactive`	Interactive mode; REPL runs and `isinteractive()` is true
`-q, --quiet`	Quiet startup: no banner, suppress REPL warnings
`--banner={yes|no|short|auto*}`	Enable or disable startup banner
`--color={yes|no|auto*}`	Enable or disable color text
`--history-file={yes*|no}`	Load or save history
`--depwarn={yes|no*|error}`	Enable or disable syntax and method deprecation warnings (`error` turns warnings into errors)
`--warn-overwrite={yes|no*}`	Enable or disable method overwrite warnings
`--warn-scope={yes*|no}`	Enable or disable warning for ambiguous top-level scope
`-C, --cpu-target <target>`	Limit usage of CPU features up to `<target>`; set to `help` to see the available options
`-O, --optimize={0|1|2*|3}`	Set the optimization level (level is 3 if `-O` is used without a level) (`$`)
`--min-`	Set the lower bound on per-module optimization

Chapter 43

The World Age mechanism

Note

World age is an advanced concept. For the vast majority of Julia users, the world age mechanism operates invisibly in the background. This documentation is intended for the few users who may encounter world-age related issues or error messages.

Julia 1.12

Prior to Julia 1.12, the world age mechanism did not apply to changes to the global binding table. The documentation in this chapter is specific to Julia 1.12+.

Warning

This manual chapter uses internal functions to introspect world age and runtime data structures as an explanatory aid. In general, unless otherwise noted the world age mechanism is not a stable interface and should be interacted with in packages through stable APIs (e.g. `invokelatest`) only. In particular, do not assume that world ages are always integers or that they have a linear order.

43.1 World age in general

The "world age counter" is a monotonically increasing counter that is incremented for every change to the global method table or the global binding table (e.g. through method definition, type definition, `import`/`using` declaration, creation of (typed) globals or definition of constants).

The current value of the global world age counter can be retrieved using the (internal) function `Base.get_world_counter`.

```
julia> Base.get_world_counter()
0x0000000000009632

julia> const x = 1

julia> Base.get_world_counter()
0x0000000000009633
```

In addition, each Task stores a local world age that determines which modifications to the global binding and method tables are currently visible to the running task. The world age of the running task will never exceed the global world age counter, but may run arbitrarily behind it. In general the term "current world age" refers to the local world age of the currently running task. The current world age may be retrieved using the (internal) function Base.tls_world_age

```
julia> function f end
f (generic function with 0 methods)

julia> begin
           @show (Int(Base.get_world_counter()), Int(Base.tls_world_age()))
           Core.eval(@__MODULE__, :(f() = 1))
           @show (Int(Base.get_world_counter()), Int(Base.tls_world_age()))
           f()
       end
(Int(Base.get_world_counter()), Int(Base.tls_world_age())) = (38452, 38452)
(Int(Base.get_world_counter()), Int(Base.tls_world_age())) = (38453, 38452)
ERROR: MethodError: no method matching f()
The applicable method may be too new: running in current world age 38452, while global world is
↪  38453.

Closest candidates are:
  f() (method too new to be called from this world context.)
   @ Main REPL[2]:3

Stacktrace:
 [1] top-level scope
   @ REPL[2]:5

julia> (f(), Int(Base.tls_world_age()))
(1, 38453)
```

Here the definition of the method f raised the global world counter, but the current world age did not change. As a result, the definition of f was not visible in the currently executing task and a MethodError resulted.

Note

The method error printing provided additional information that `f()` is available in a newer world age. This information is added by the error display, not the task that threw the `MethodError`. The thrown `MethodError` is identical whether or not a matching definition of `f()` exists in a newer world age.

However, note that the definition of `f()` was subsequently available at the next REPL prompt, because the current task's world age had been raised. In general, certain syntactic constructs (in particular most definitions) will raise the current task's world age to the latest global world age, thus making all changes (both from the current task and any concurrently executing other tasks) visible. The following statements raise the current world age:

1. An explicit invocation of `Core.@latestworld`
2. The start of every top-level statement

3. The start of every REPL prompt
4. Any type or struct definition
5. Any method definition
6. Any constant declaration
7. Any global variable declaration (but not a global variable assignment)
8. Any `using`, `import`, `export` or `public` statement
9. Certain other macros like `@eval` (depends on the macro implementation)

Note, however, that the current task's world age may only ever be permanently incremented at top level. As a general rule, using any of the above statements in non-top-level scope is a syntax error:

```
julia> f() = Core.@latestworld
ERROR: syntax: World age increment not at top level
Stacktrace:
 [1] top-level scope
   @ REPL[5]:1
```

When it isn't (for example for `@eval`), the world age side effect is ignored.

As a result of these rules, Julia may assume that the world age does not change within the execution of an ordinary function.

```
function my_function()
    before = Base.tls_world_age()
    # Any arbitrary code
    after = Base.tls_world_age()
    @assert before === after # always true
end
```

This is the key invariant that allows Julia to optimize based on the current state of its global data structures, while still having the well-defined ability to change these data structures.

43.2 Temporarily raising the world age using `invokelatest`

As described above, it is not possible to permanently raise the world age for the remainder of a `Task`'s execution unless the task is executing top-level statements. However, it is possible to temporarily change the world age in a scoped manner using `invokelatest`:

```
julia> function f end
f (generic function with 0 methods)

julia> begin
           Core.eval(@__MODULE__, :(f() = 1))
           invokelatest(f)
       end
1
```

`invokelatest` will temporarily raise the current task's world age to the latest global world age (at entry to `invokelatest`) and execute the provided function. Note that the world age will return to its prior value upon exit from `invokelatest`.

43.3 World age and const struct redefinitions

The semantics described above for method redefinition also apply to redefinition of constants:

```
julia> const x = 1
1

julia> get_const() = x
get_const (generic function with 1 method)

julia> begin
           @show get_const()
           Core.eval(@__MODULE__, :(const x = 2))
           @show get_const()
           Core.@latestworld
           @show get_const()
       end
get_const() = 1
get_const() = 1
get_const() = 2
2
```

However, for the avoidance of doubt, they do not apply to ordinary assignment to global variables, which becomes visible immediately:

```
julia> global y = 1
1

julia> get_global() = y
get_global (generic function with 1 method)

julia> begin
           @show get_global()
           Core.eval(@__MODULE__, :(y = 2))
           @show get_global()
       end
get_global() = 1
get_global() = 2
2
```

One particular special case of constant reassignment is the redefinition of struct types:

```
julia> struct MyStruct
           x::Int
       end
```

```
julia> const one_field = MyStruct(1)
MyStruct(1)

julia> struct MyStruct
           x::Int
           y::Float64
       end

julia> const two_field = MyStruct(1, 2.0)
MyStruct(1, 2.0)

julia> one_field
@world(MyStruct, 38452:38455)(1)

julia> two_field
MyStruct(1, 2.0)
```

Internally the two definitions of `MyStruct` are entirely separate types. However, after the new `MyStruct` type is defined, there is no longer any default binding for the original definition of `MyStruct`. To nevertheless facilitate access to these types, the special `@world` macro may be used to access the meaning of a name in a previous world. However, this facility is intended for introspection only and in particular note that world age numbers are not stable across precompilation and should in general be treated opaquely.

Binding partition introspection

In certain cases, it can be helpful to introspect the system's understanding of what a binding means in any particular world age. The default display printing of `Core.Binding` provides a helpful summary (e.g. on the `MyStruct` example from above):

```
julia> convert(Core.Binding, GlobalRef(@__MODULE__, :MyStruct))
Binding Main.MyStruct
   38456:∞ - constant binding to MyStruct
   38452:38455 - constant binding to @world(MyStruct, 38452:38455)
   38451:38451 - backdated constant binding to @world(MyStruct, 38452:38455)
   0:38450 - backdated constant binding to @world(MyStruct, 38452:38455)
```

43.4 World age and `using`/`import`

Bindings provided via `using` and `import` also operate via the world age mechanism. Binding resolution is a stateless function of the `import` and `using` definitions visible in the current world age. For example:

```
julia> module M1; const x = 1; export x; end

julia> module M2; const x = 2; export x; end

julia> using .M1

julia> x
1

julia> using .M2
```

```
julia> x
ERROR: UndefVarError: `x` not defined in `Main`
Hint: It looks like two or more modules export different bindings with this name, resulting in
↪  ambiguity. Try explicitly importing it from a particular module, or qualifying the name with
↪  the module it should come from.

julia> convert(Core.Binding, GlobalRef(@__MODULE__, :x))
Binding Main.x
   38458:∞ - ambiguous binding - guard entry
   38457:38457 - implicit `using` resolved to constant 1
```

43.5 World age capture

Certain language features capture the current task's world age. Perhaps the most common of these is creation of new tasks. Newly created tasks will inherit the creating task's local world age at creation time and will retain said world age (unless explicitly raised) even if the originating tasks raises its world age:

```
julia> const x = 1

julia> t = @task (wait(); println("Running now"); x);

julia> const x = 2

julia> schedule(t);
Running now

julia> x
2

julia> fetch(t)
1
```

In addition to tasks, opaque closures also capture their world age at creation. See `Base.Experimental.@opaque`.

`Base.@world` – Macro.

```
@world(sym, world)
```

Resolve the binding `sym` in world `world`. See `invoke_in_world` for running arbitrary code in fixed worlds. `world` may be `UnitRange`, in which case the macro will error unless the binding is valid and has the same value across the entire world range.

As a special case, the world ∞ always refers to the latest world, even if that world is newer than the world currently running.

The `@world` macro is primarily used in the printing of bindings that are no longer available in the current world.

Example

```
julia> struct Foo; a::Int; end
Foo
```

```
julia> fold = Foo(1)

julia> Int(Base.get_world_counter())
26866

julia> struct Foo; a::Int; b::Int end
Foo

julia> fold
@world(Foo, 26866)(1)
```

Julia 1.12

This functionality requires at least Julia 1.12.

source

Base.get_world_counter – Function.

```
get_world_counter()
```

Returns the current maximum world-age counter. This counter is monotonically increasing.

Warning

This counter is global and may change at any time between invocations. In general, most reflection functions operate on the current task's world age, rather than the global maximum world age. See tls_world_age as well as the manual chapter of world age.

source

Base.tls_world_age – Function.

```
tls_world_age()
```

Returns the world the current_task() is executing within.

source

Base.invoke_in_world – Function.

```
invoke_in_world(world, f, args...; kwargs...)
```

Call `f(args...; kwargs...)` in a fixed world age, `world`.

This is useful for infrastructure running in the user's Julia session which is not part of the user's program. For example, things related to the REPL, editor support libraries, etc. In these cases it can be useful to prevent unwanted method invalidation and recompilation latency, and to prevent the user from breaking supporting infrastructure by mistake.

The global world age can be queried using Base.get_world_counter() and stored for later use within the lifetime of the current Julia session, or when serializing and reloading the system image.

Technically, `invoke_in_world` will prevent any function called by `f` from being extended by the user during their Julia session. That is, generic function method tables seen by `f` (and any functions it calls) will be frozen as they existed at the given `world` age. In a sense, this is like the opposite of `invokelatest`.

Note

It is not valid to store world ages obtained in precompilation for later use. This is because precompilation generates a "parallel universe" where the world age refers to system state unrelated to the main Julia session.

source

`Base.Experimental.@opaque` – Macro.

```
@opaque ([type, ]args...) -> body
```

Marks a given closure as "opaque". Opaque closures capture the world age of their creation (as opposed to their invocation). This allows for more aggressive optimization of the capture list, but trades off against the ability to inline opaque closures at the call site, if their creation is not statically visible.

An argument tuple type (`type`) may optionally be specified, to specify allowed argument types in a more flexible way. In particular, the argument type may be fixed length even if the function is variadic.

Warning

This interface is experimental and subject to change or removal without notice.

source

Part II

Base

Chapter 44

Essentials

44.1 Introduction

Julia Base contains a range of functions and macros appropriate for performing scientific and numerical computing, but is also as broad as those of many general-purpose programming languages. Additional functionality is available from a growing collection of available packages. Functions are grouped by topic below.

Some general notes:

- To use module functions, use `import Module` to import the module, and `Module.fn(x)` to use the functions.
- Alternatively, `using Module` will import all exported `Module` functions into the current namespace.
- By convention, function names ending with an exclamation point (`!`) modify their arguments. Some functions have both modifying (e.g., `sort!`) and non-modifying (`sort`) versions.

The behaviors of Base and standard libraries are stable as defined in SemVer only if they are documented; i.e., included in the Julia documentation and not marked as unstable. See API FAQ for more information.

44.2 Getting Around

`Base.exit` – Function.

```
exit(code=0)
```

Stop the program with an exit code. The default exit code is zero, indicating that the program completed successfully. In an interactive session, `exit()` can be called with the keyboard shortcut `^D`.

source

`Base.atexit` – Function.

```
atexit(f)
```

Register a zero- or one-argument function `f()` to be called at process exit. `atexit()` hooks are called in last in first out (LIFO) order and run before object finalizers.

If `f` has a method defined for one integer argument, it will be called as `f(n::Int32)`, where `n` is the current exit code, otherwise it will be called as `f()`.

Julia 1.9

The one-argument form requires Julia 1.9

Exit hooks are allowed to call `exit(n)`, in which case Julia will exit with exit code `n` (instead of the original exit code). If more than one exit hook calls `exit(n)`, then Julia will exit with the exit code corresponding to the last called exit hook that calls `exit(n)`. (Because exit hooks are called in LIFO order, "last called" is equivalent to "first registered".)

Note: Once all exit hooks have been called, no more exit hooks can be registered, and any call to `atexit(f)` after all hooks have completed will throw an exception. This situation may occur if you are registering exit hooks from background Tasks that may still be executing concurrently during shutdown.

source

`Base.isinteractive` – Function.

```
isinteractive()::Bool
```

Determine whether Julia is running an interactive session.

source

`Base.summarysize` – Function.

```
Base.summarysize(obj; count = false, exclude=Union{...}, chargeall=Union{...})::Int
```

Compute all unique objects reachable from the argument and return either their size in memory (in bytes) or the number of allocations they span.

Keyword Arguments

- `count`: if false, return the total size of the objects in memory. if true, return the number of allocations spanned by the object.
- `exclude`: specifies the types of objects to exclude from the traversal.
- `chargeall`: specifies the types of objects to always charge the size of all of their fields, even if those fields would normally be excluded.

See also `sizeof`.

Examples

```
julia> Base.summarysize(1.0)
8

julia> Base.summarysize(Ref(rand(100)))
848
```

```
julia> sizeof(Ref(rand(100)))
8

julia> Base.summarysize(Core.svec(1.0, "testing", true); count=true)
4
```

source

Base.__precompile__ – Function.

```
__precompile__(isprecompilable::Bool)
```

Specify whether the file calling this function is precompilable, defaulting to `true`. If a module or file is *not* safely precompilable, it should call `__precompile__(false)` in order to throw an error if Julia attempts to precompile it.

source

Base.include – Function.

```
Base.include([mapexpr::Function,] m::Module, path::AbstractString)
```

Evaluate the contents of the input source file in the global scope of module `m`. Every module (except those defined with `baremodule`) has its own definition of `include` omitting the `m` argument, which evaluates the file in that module. Returns the result of the last evaluated expression of the input file. During including, a task-local include path is set to the directory containing the file. Nested calls to `include` will search relative to that path. This function is typically used to load source interactively, or to combine files in packages that are broken into multiple source files.

The optional first argument `mapexpr` can be used to transform the included code before it is evaluated: for each parsed expression `expr` in `path`, the `include` function actually evaluates `mapexpr(expr)`. If it is omitted, `mapexpr` defaults to `identity`.

Julia 1.5

Julia 1.5 is required for passing the `mapexpr` argument.

source

include – Function.

```
include([mapexpr::Function,] path::AbstractString)
```

Evaluate the contents of the input source file in the global scope of the containing module. Every `Module` (except those defined with `baremodule`) has a private 1-argument definition of `include`, which evaluates the file in that module, for use inside that module. Returns the result of the last evaluated expression of the input file. During including, a task-local include path is set to the directory containing the file. Nested calls to `include` will search relative to that path. This function is typically used to load source interactively, or to combine files in packages that are broken into multiple source files. The argument `path` is normalized using `normpath` which will resolve relative path tokens such as `..` and convert `/` to the appropriate path separator.

The optional first argument `mapexpr` can be used to transform the included code before it is evaluated: for each parsed expression `expr` in `path`, the `include` function actually evaluates `mapexpr(expr)`. If it is omitted, `mapexpr` defaults to `identity`.

Use `Base.include` to evaluate a file into another module.

Note

Julia's syntax lowering recognizes an explicit call to a literal `include` at top-level and inserts an implicit `@Core.latestworld` to make any include'd definitions visible to subsequent code. Note however that this recognition is *syntactic*. I.e. assigning `const myinclude = include` may require an explicit `@Core.latestworld` call after `myinclude`.

Julia 1.5

Julia 1.5 is required for passing the `mapexpr` argument.

source

`Base.include_string` – Function.

```
include_string([mapexpr::Function,] m::Module, code::AbstractString,
↪  filename::AbstractString="string")
```

Like `include`, except reads code from the given string rather than from a file.

The optional first argument `mapexpr` can be used to transform the included code before it is evaluated: for each parsed expression `expr` in `code`, the `include_string` function actually evaluates `mapexpr(expr)`. If it is omitted, `mapexpr` defaults to `identity`.

Julia 1.5

Julia 1.5 is required for passing the `mapexpr` argument.

source

`Base.include_dependency` – Function.

```
include_dependency(path::AbstractString; track_content::Bool=true)
```

In a module, declare that the file, directory, or symbolic link specified by `path` (relative or absolute) is a dependency for precompilation; that is, if `track_content=true` the module will need to be recompiled if the content of `path` changes (if `path` is a directory the content equals `join(readdir(path))`). If `track_content=false` recompilation is triggered when the modification time `mtime` of `path` changes.

This is only needed if your module depends on a path that is not used via `include`. It has no effect outside of compilation.

Julia 1.11

Keyword argument `track_content` requires at least Julia 1.11. An error is now thrown if `path` is not readable.

source

`__init__` – Keyword.

```
__init__
```

The `__init__()` function in a module executes immediately *after* the module is loaded at runtime for the first time. It is called once, after all other statements in the module have been executed. Because it is called after fully importing the module, `__init__` functions of submodules will be executed first. Two typical uses of `__init__` are calling runtime initialization functions of external C libraries and initializing global constants that involve pointers returned by external libraries. See the manual section about modules for more details.

See also: `OncePerProcess`.

Examples

```
const foo_data_ptr = Ref{Ptr{Cvoid}}(0)
function __init__()
    ccall((:foo_init, :libfoo), Cvoid, ())
    foo_data_ptr[] = ccall((:foo_data, :libfoo), Ptr{Cvoid}, ())
    nothing
end
```

source

Base.OncePerProcess – Type.

```
OncePerProcess{T}(init::Function)() -> T
```

Calling a `OncePerProcess` object returns a value of type T by running the function `initializer` exactly once per process. All concurrent and future calls in the same process will return exactly the same value. This is useful in code that will be precompiled, as it allows setting up caches or other state which won't get serialized.

Julia 1.12

This type requires Julia 1.12 or later.

Example

```
julia> const global_state = Base.OncePerProcess{Vector{UInt32}}() do
           println("Making lazy global value...done.")
           return [Libc.rand()]
       end;

julia> (procstate = global_state()) |> typeof
Making lazy global value...done.
Vector{UInt32} (alias for Array{UInt32, 1})

julia> procstate === global_state()
true

julia> procstate === fetch(@async global_state())
true
```

source

Base.OncePerTask – Type.

```
OncePerTask{T}(init::Function)() -> T
```

Calling a OncePerTask object returns a value of type T by running the function `initializer` exactly once per Task. All future calls in the same Task will return exactly the same value.

See also: `task_local_storage`.

Julia 1.12

This type requires Julia 1.12 or later.

Example

```
julia> const task_state = Base.OncePerTask{Vector{UInt32}}() do
           println("Making lazy task value...done.")
           return [Libc.rand()]
       end;

julia> (taskvec = task_state()) |> typeof
Making lazy task value...done.
Vector{UInt32} (alias for Array{UInt32, 1})

julia> taskvec === task_state()
true

julia> taskvec === fetch(@async task_state())
Making lazy task value...done.
false
```

source

Base.OncePerThread – Type.

```
OncePerThread{T}(init::Function)() -> T
```

Calling a OncePerThread object returns a value of type T by running the function `initializer` exactly once per thread. All future calls in the same thread, and concurrent or future calls with the same thread id, will return exactly the same value. The object can also be indexed by the threadid for any existing thread, to get (or initialize *on this thread*) the value stored for that thread. Incorrect usage can lead to data-races or memory corruption so use only if that behavior is correct within your library's threading-safety design.

Warning

It is not necessarily true that a Task only runs on one thread, therefore the value returned here may alias other values or change in the middle of your program. This function may get deprecated in the future. If initializer yields, the thread running the current task after the call might not be the same as the one at the start of the call.

See also: `OncePerTask`.

Julia 1.12

This type requires Julia 1.12 or later.

Example

```
julia> const thread_state = Base.OncePerThread{Vector{UInt32}}() do
           println("Making lazy thread value...done.")
           return [Libc.rand()]
       end;

julia> (threadvec = thread_state()) |> typeof
Making lazy thread value...done.
Vector{UInt32} (alias for Array{UInt32, 1})

julia> threadvec === fetch(@async thread_state())
true

julia> threadvec === thread_state[Threads.threadid()]
true
```

source

Base.which – Method.

```
which(f, types)
```

Returns the method of f (a Method object) that would be called for arguments of the given types.

If types is an abstract type, then the method that would be called by invoke is returned.

See also: parentmodule, @which, and @edit.

source

Base.methods – Function.

```
methods(f, [types], [module])
```

Return the method table for f.

If types is specified, return an array of methods whose types match. If module is specified, return an array of methods defined in that module. A list of modules can also be specified as an array or set.

Julia 1.4

At least Julia 1.4 is required for specifying a module.

See also: which, @which and methodswith.

source

Base.@show – Macro.

```
@show exs...
```

Prints one or more expressions, and their results, to stdout, and returns the last result.

See also: show, @info, println.

Examples

```
julia> x = @show 1+2
1 + 2 = 3
3

julia> @show x^2 x/2;
x ^ 2 = 9
x / 2 = 1.5
```

source

Base.MainInclude.ans – Constant.

```
ans
```

A variable referring to the last computed value, automatically imported to the interactive prompt.

source

Base.MainInclude.err – Constant.

```
err
```

A variable referring to the last thrown errors, automatically imported to the interactive prompt. The thrown errors are collected in a stack of exceptions.

source

Base.active_project – Function.

```
active_project()
```

Return the path of the active `Project.toml` file. See also `Base.set_active_project`.

source

Base.set_active_project – Function.

```
set_active_project(projfile::Union{AbstractString,Nothing})
```

Set the active `Project.toml` file to `projfile`. See also `Base.active_project`.

Julia 1.8

This function requires at least Julia 1.8.

source

Base.active_manifest – Function.

```
active_manifest()
active_manifest(project_file::AbstractString)
```

Return the path of the active manifest file, or the manifest file that would be used for a given `project_file`.

In a stacked environment (where multiple environments exist in the load path), this returns the manifest file for the primary (active) environment only, not the manifests from other environments in the stack. See the manual section on Environment stacks for more details on how stacked environments work.

See Project environments for details on the difference between a project and a manifest, and the naming options and their priority in package loading.

See also Base.active_project, Base.set_active_project.

Julia 1.13

This function requires at least Julia 1.13.

source

44.3 Keywords

This is the list of reserved keywords in Julia: baremodule, begin, break, catch, const, continue, do, else, elseif, end, export, false, finally, for, function, global, if, import, let, local, macro, module, quote, return, struct, true, try, using, while. Those keywords are not allowed to be used as variable names.

The following two-word sequences are reserved: abstract type, mutable struct, primitive type. However, you can create variables with names: abstract, mutable, primitive and type.

Finally: where is parsed as an infix operator for writing parametric method and type definitions; in and isa are parsed as infix operators; public is parsed as a keyword when beginning a toplevel statement; outer is parsed as a keyword when used to modify the scope of a variable in an iteration specification of a for loop; and as is used as a keyword to rename an identifier brought into scope by import or using. Creation of variables named where, in, isa, outer and as is allowed, though.

module – Keyword.

```
module
```

module declares a Module, which is a separate global variable workspace. Within a module, you can control which names from other modules are visible (via importing), and specify which of your names are intended to be public (via export and public). Modules allow you to create top-level definitions without worrying about name conflicts when your code is used together with somebody else's. See the manual section about modules for more details.

Examples

```
module Foo
import Base.show
export MyType, foo

struct MyType
    x
end

bar(x) = 2x
foo(a::MyType) = bar(a.x) + 1
show(io::IO, a::MyType) = print(io, "MyType $(a.x)")
end
```

source

`export` – Keyword.

```
export
```

`export` is used within modules to tell Julia which names should be made available to the user. For example: `export foo` makes the name `foo` available when `using` the module. See the manual section about modules for details.

source

`public` – Keyword.

```
public
```

`public` is used within modules to tell Julia which names are part of the public API of the module. For example: `public foo` indicates that the name `foo` is public, without making it available when `using` the module.

As `export` already indicates that a name is public, it is unnecessary and an error to declare a name both as `public` and as `exported`. See the manual section about modules for details.

Julia 1.11

The public keyword was added in Julia 1.11. Prior to this the notion of publicness was less explicit.

source

`import` – Keyword.

```
import
```

`import Foo` will load the module or package `Foo`. Names from the imported `Foo` module can be accessed with dot syntax (e.g. `Foo.foo` to access the name `foo`). See the manual section about modules for details.

source

`using` – Keyword.

```
using
```

`using Foo` will load the module or package `Foo` and make its `export`ed names available for direct use. Names can also be used via dot syntax (e.g. `Foo.foo` to access the name `foo`), whether they are `exported` or not. See the manual section about modules for details.

Note

When two or more packages/modules export a name and that name does not refer to the same thing in each of the packages, and the packages are loaded via `using` without an explicit list of names, it is an error to reference that name without qualification. It is thus recommended that code intended to be forward-compatible with future versions of its dependencies and of Julia, e.g., code in released packages, list the names it uses from each loaded package, e.g., `using Foo: Foo, f` rather than `using Foo`.

source

`as` – Keyword.

```
as
```

`as` is used as a keyword to rename an identifier brought into scope by `import` or `using`, for the purpose of working around name conflicts as well as for shortening names. (Outside of `import` or `using` statements, `as` is not a keyword and can be used as an ordinary identifier.)

`import LinearAlgebra as LA` brings the imported `LinearAlgebra` standard library into scope as `LA`.

`import LinearAlgebra: eigen as eig, cholesky as chol` brings the `eigen` and `cholesky` methods from `LinearAlgebra` into scope as `eig` and `chol` respectively.

`as` works with `using` only when individual identifiers are brought into scope. For example, `using LinearAlgebra: eigen as eig` or `using LinearAlgebra: eigen as eig, cholesky as chol` works, but `using LinearAlgebra as LA` is invalid syntax, since it is nonsensical to rename *all* exported names from `LinearAlgebra` to `LA`.

source

`baremodule` – Keyword.

```
baremodule
```

`baremodule` declares a module that does not contain `using Base` or local definitions of `eval` and `include`. It does still import `Core`. In other words,

```
module Mod

...

end
```

is equivalent to

```
baremodule Mod

using Base

eval(x) = Core.eval(Mod, x)
include(p) = Base.include(Mod, p)

...

end
```

source

`function` – Keyword.

```
function
```

Functions are defined with the `function` keyword:

```
function add(a, b)
    return a + b
end
```

Or the short form notation:

```
add(a, b) = a + b
```

The use of the `return` keyword is exactly the same as in other languages, but is often optional. A function without an explicit `return` statement will return the last expression in the function body.

source

`macro` – Keyword.

```
macro
```

`macro` defines a method for inserting generated code into a program. A macro maps a sequence of argument expressions to a returned expression, and the resulting expression is substituted directly into the program at the point where the macro is invoked. Macros are a way to run generated code without calling `eval`, since the generated code instead simply becomes part of the surrounding program. Macro arguments may include expressions, literal values, and symbols. Macros can be defined for variable number of arguments (varargs), but do not accept keyword arguments. Every macro also implicitly gets passed the arguments `__source__`, which contains the line number and file name the macro is called from, and `__module__`, which is the module the macro is expanded in.

See the manual section on Metaprogramming for more information about how to write a macro.

Examples

```
julia> macro sayhello(name)
           return :( println("Hello, ", $name, "!") )
       end
@sayhello (macro with 1 method)

julia> @sayhello "Charlie"
Hello, Charlie!

julia> macro saylots(x...)
           return :( println("Say: ", $(x...)) )
       end
@saylots (macro with 1 method)

julia> @saylots "hey " "there " "friend"
Say: hey there friend
```

source

`return` – Keyword.

```
return
```

`return x` causes the enclosing function to exit early, passing the given value `x` back to its caller. `return` by itself with no value is equivalent to `return nothing` (see `nothing`).

```
function compare(a, b)
    a == b && return "equal to"
    a < b ? "less than" : "greater than"
end
```

In general you can place a `return` statement anywhere within a function body, including within deeply nested loops or conditionals, but be careful with `do` blocks. For example:

```
function test1(xs)
    for x in xs
        iseven(x) && return 2x
    end
end

function test2(xs)
    map(xs) do x
        iseven(x) && return 2x
        x
    end
end
```

In the first example, the return breaks out of `test1` as soon as it hits an even number, so `test1([5,6,7])` returns 12.

You might expect the second example to behave the same way, but in fact the `return` there only breaks out of the *inner* function (inside the `do` block) and gives a value back to `map`. `test2([5,6,7])` then returns `[5,12,7]`.

When used in a top-level expression (i.e. outside any function), `return` causes the entire current top-level expression to terminate early.

source

`do` – Keyword.

```
do
```

Create an anonymous function and pass it as the first argument to a function call. For example:

```
map(1:10) do x
    2x
end
```

is equivalent to `map(x->2x, 1:10)`.

Use multiple arguments like so:

```
map(1:10, 11:20) do x, y
    x + y
end
```

source

`begin` – Keyword.

```
begin
```

begin...end denotes a block of code.

```
begin
    println("Hello, ")
    println("World!")
end
```

Usually begin will not be necessary, since keywords such as function and let implicitly begin blocks of code. See also ;.

begin may also be used when indexing to represent the first index of a collection or the first index of a dimension of an array. For example, a[begin] is the first element of an array a.

Julia 1.4

Use of begin as an index requires Julia 1.4 or later.

Examples

```
julia> A = [1 2; 3 4]
2×2 Matrix{Int64}:
 1  2
 3  4

julia> A[begin, :]
2-element Matrix{Int64}:
 1
 2
```

source

end – Keyword.

```
end
```

end marks the conclusion of a block of expressions, for example module, struct, mutable struct, begin, let, for etc.

end may also be used when indexing to represent the last index of a collection or the last index of a dimension of an array.

Examples

```
julia> A = [1 2; 3 4]
2×2 Matrix{Int64}:
 1  2
 3  4

julia> A[end, :]
2-element Vector{Int64}:
 3
 4
```

source

`let` – Keyword.

```
let
```

`let` blocks create a new hard scope and optionally introduce new local bindings.

Just like the other scope constructs, `let` blocks define the block of code where newly introduced local variables are accessible. Additionally, the syntax has a special meaning for comma-separated assignments and variable names that may optionally appear on the same line as the `let`:

```
let var1 = value1, var2, var3 = value3
    code
end
```

The variables introduced on this line are local to the `let` block and the assignments are evaluated in order, with each right-hand side evaluated in the scope without considering the name on the left-hand side. Therefore it makes sense to write something like `let x = x`, since the two x variables are distinct with the left-hand side locally shadowing the x from the outer scope. This can even be a useful idiom as new local variables are freshly created each time local scopes are entered, but this is only observable in the case of variables that outlive their scope via closures. A `let` variable without an assignment, such as `var2` in the example above, declares a new local variable that is not yet bound to a value.

By contrast, `begin` blocks also group multiple expressions together but do not introduce scope or have the special assignment syntax.

Examples

In the function below, there is a single x that is iteratively updated three times by the `map`. The closures returned all reference that one x at its final value:

```
julia> function test_outer_x()
           x = 0
           map(1:3) do _
               x += 1
               return ()->x
           end
       end
test_outer_x (generic function with 1 method)

julia> [f() for f in test_outer_x()]
3-element Vector{Int64}:
 3
 3
 3
```

If, however, we add a `let` block that introduces a *new* local variable we will end up with three distinct variables being captured (one at each iteration) even though we chose to use (shadow) the same name.

```
julia> function test_let_x()
           x = 0
           map(1:3) do _
               x += 1
```

```
            let x = x
                return ()->x
            end
        end
    end
test_let_x (generic function with 1 method)

julia> [f() for f in test_let_x()]
3-element Vector{Int64}:
 1
 2
 3
```

All scope constructs that introduce new local variables behave this way when repeatedly run; the distinctive feature of `let` is its ability to succinctly declare new `locals` that may shadow outer variables of the same name. For example, directly using the argument of the do function similarly captures three distinct variables:

```
julia> function test_do_x()
           map(1:3) do x
               return ()->x
           end
       end
test_do_x (generic function with 1 method)

julia> [f() for f in test_do_x()]
3-element Vector{Int64}:
 1
 2
 3
```

source

`if` – Keyword.

```
if/elseif/else
```

`if`/`elseif`/`else` performs conditional evaluation, which allows portions of code to be evaluated or not evaluated depending on the value of a boolean expression. Here is the anatomy of the `if`/`elseif`/`else` conditional syntax:

```
if x < y
    println("x is less than y")
elseif x > y
    println("x is greater than y")
else
    println("x is equal to y")
end
```

If the condition expression `x < y` is true, then the corresponding block is evaluated; otherwise the condition expression `x > y` is evaluated, and if it is true, the corresponding block is evaluated; if neither expression is true, the `else` block is evaluated. The `elseif` and `else` blocks are optional, and as many `elseif` blocks as desired can be used.

In contrast to some other languages conditions must be of type `Bool`. It does not suffice for conditions to be convertible to `Bool`.

```
julia> if 1 end
ERROR: TypeError: non-boolean (Int64) used in boolean context
```

source

`for` – Keyword.

```
for
```

`for` loops repeatedly evaluate a block of statements while iterating over a sequence of values.

The iteration variable is always a new variable, even if a variable of the same name exists in the enclosing scope. Use `outer` to reuse an existing local variable for iteration.

Examples

```
julia> for i in [1, 4, 0]
           println(i)
       end
1
4
0
```

source

`while` – Keyword.

```
while
```

`while` loops repeatedly evaluate a conditional expression, and continue evaluating the body of the while loop as long as the expression remains true. If the condition expression is false when the while loop is first reached, the body is never evaluated.

Examples

```
julia> i = 1
1

julia> while i < 5
           println(i)
           global i += 1
       end
1
2
3
4
```

source

`break` – Keyword.

```
break
```

Break out of a loop immediately.

Examples

```
julia> i = 0
0

julia> while true
           global i += 1
           i > 5 && break
           println(i)
       end
1
2
3
4
5
```

source

`continue` – Keyword.

```
continue
```

Skip the rest of the current loop iteration.

Examples

```
julia> for i = 1:6
           iseven(i) && continue
           println(i)
       end
1
3
5
```

source

`try` – Keyword.

```
try/catch
```

A `try/catch` statement allows intercepting errors (exceptions) thrown by `throw` so that program execution can continue. For example, the following code attempts to write a file, but warns the user and proceeds instead of terminating execution if the file cannot be written:

```
try
    open("/danger", "w") do f
        println(f, "Hello")
    end
catch
    @warn "Could not write file."
end
```

or, when the file cannot be read into a variable:

```
lines = try
    open("/danger", "r") do f
        readlines(f)
    end
catch
    @warn "File not found."
end
```

The syntax `catch e` (where `e` is any variable) assigns the thrown exception object to the given variable within the `catch` block.

```
try
    a_dangerous_operation()
catch e
    if isa(e, EOFError)
        @warn "The operation failed - EOF."
    elseif isa(e, OutOfMemoryError)
        @warn "The operation failed - OOM."
    else
        rethrow() # ensure other exceptions can bubble up the call stack
    end
end
```

The power of the `try`/`catch` construct lies in the ability to unwind a deeply nested computation immediately to a much higher level in the stack of calling functions.

A `try/catch` block can also have an `else` clause that executes only if no exception occurred:

```
try
    a_dangerous_operation()
catch
    @warn "The operation failed."
else
    @info "The operation succeeded."
end
```

A `try` or `try/catch` block can also have a `finally` clause that executes at the end, regardless of whether an exception occurred. For example, this can be used to guarantee that an opened file is closed:

```
f = open("file")
try
    operate_on_file(f)
catch
    @warn "An error occurred!"
finally
    close(f)
end
```

(`finally` can also be used without a `catch` block.)

Julia 1.8

Else clauses require at least Julia 1.8.

source

`finally` – Keyword.

```
finally
```

Run some code when a given `try` block of code exits, regardless of how it exits. For example, here is how we can guarantee that an opened file is closed:

```
f = open("file")
try
    operate_on_file(f)
finally
    close(f)
end
```

When control leaves the `try` block (for example, due to a `return`, or just finishing normally), `close(f)` will be executed. If the `try` block exits due to an exception, the exception will continue propagating. A `catch` block may be combined with `try` and `finally` as well. In this case the `finally` block will run after `catch` has handled the error.

When evaluating a `try/catch/else/finally` expression, the value of the entire expression is the value of the last block executed, excluding the `finally` block. For example:

```
julia> try
           1
       finally
           2
       end
1

julia> try
           error("")
       catch
           1
       else
           2
       finally
           3
       end
1

julia> try
           0
       catch
           1
       else
           2
       finally
```

```
            3
        end
2
```

source

`quote` – Keyword.

```
quote
```

`quote` creates multiple expression objects in a block without using the explicit `Expr` constructor. For example:

```
ex = quote
    x = 1
    y = 2
    x + y
end
```

Unlike the other means of quoting, `:( ... )`, this form introduces `QuoteNode` elements to the expression tree, which must be considered when directly manipulating the tree. For other purposes, `:( ... )` and `quote .. end` blocks are treated identically.

source

`local` – Keyword.

```
local
```

`local` introduces a new local variable. See the manual section on variable scoping for more information.

Examples

```
julia> function foo(n)
            x = 0
            for i = 1:n
                local x # introduce a loop-local x
                x = i
            end
            x
        end
foo (generic function with 1 method)

julia> foo(10)
0
```

source

`global` – Keyword.

```
global
```

`global x` makes x in the current scope and its inner scopes refer to the global variable of that name. See the manual section on variable scoping for more information.

Examples

```
julia> z = 3
3

julia> function foo()
           global z = 6 # use the z variable defined outside foo
       end
foo (generic function with 1 method)

julia> foo()
6

julia> z
6
```

source

`outer` – Keyword.

```
for outer
```

Reuse an existing local variable for iteration in a `for` loop.

See the manual section on variable scoping for more information.

See also `for`.

Examples

```
julia> function f()
           i = 0
           for i = 1:3
               # empty
           end
           return i
       end;

julia> f()
0
```

```
julia> function f()
           i = 0
           for outer i = 1:3
               # empty
           end
           return i
       end;

julia> f()
3
```

```
julia> i = 0 # global variable
       for outer i = 1:3
       end
ERROR: syntax: no outer local variable declaration exists for "for outer"
[...]
```

source

`const` – Keyword.

```
const
```

`const` is used to declare global variables whose values will not change. In almost all code (and particularly performance sensitive code) global variables should be declared constant in this way.

```
const x = 5
```

Multiple variables can be declared within a single `const`:

```
const y, z = 7, 11
```

Note that `const` only applies to one `=` operation, therefore `const x = y = 1` declares `x` to be constant but not `y`. On the other hand, `const x = const y = 1` declares both `x` and `y` constant.

Note that "constant-ness" does not extend into mutable containers; only the association between a variable and its value is constant. If `x` is an array or dictionary (for example) you can still modify, add, or remove elements.

In some cases changing the value of a `const` variable gives a warning instead of an error. However, this can produce unpredictable behavior or corrupt the state of your program, and so should be avoided. This feature is intended only for convenience during interactive use.

source

`struct` – Keyword.

```
struct
```

The most commonly used kind of type in Julia is a struct, specified as a name and a set of fields.

```
struct Point
    x
    y
end
```

Fields can have type restrictions, which may be parameterized:

```
struct Point{X}
    x::X
    y::Float64
end
```

A struct can also declare an abstract super type via `<:` syntax:

```
struct Point <: AbstractPoint
    x
    y
end
```

`structs` are immutable by default; an instance of one of these types cannot be modified after construction. Use `mutable struct` instead to declare a type whose instances can be modified.

See the manual section on Composite Types for more details, such as how to define constructors.

source

`mutable struct` – Keyword.

```
mutable struct
```

`mutable struct` is similar to `struct`, but additionally allows the fields of the type to be set after construction.

Individual fields of a mutable struct can be marked as `const` to make them immutable:

```
mutable struct Baz
    a::Int
    const b::Float64
end
```

Julia 1.8

The `const` keyword for fields of mutable structs requires at least Julia 1.8.

See the manual section on Composite Types for more information.

source

`Base.@kwdef` – Macro.

```
@kwdef typedef
```

This is a helper macro that automatically defines a keyword-based constructor for the type declared in the expression `typedef`, which must be a `struct` or `mutable struct` expression. The default argument is supplied by declaring fields of the form `field::T = default` or `field = default`. If no default is provided then the keyword argument becomes a required keyword argument in the resulting type constructor.

Inner constructors can still be defined, but at least one should accept arguments in the same form as the default inner constructor (i.e. one positional argument per field) in order to function correctly with the keyword outer constructor.

Julia 1.1

`Base.@kwdef` for parametric structs, and structs with supertypes requires at least Julia 1.1.

Julia 1.9

This macro is exported as of Julia 1.9.

Examples

```
julia> @kwdef struct Foo
           a::Int = 1         # specified default
           b::String          # required keyword
       end
Foo

julia> Foo(b="hi")
Foo(1, "hi")

julia> Foo()
ERROR: UndefKeywordError: keyword argument `b` not assigned
Stacktrace:
[...]
```

source

`abstract type` – Keyword.

```
abstract type
```

`abstract type` declares a type that cannot be instantiated, and serves only as a node in the type graph, thereby describing sets of related concrete types: those concrete types which are their descendants. Abstract types form the conceptual hierarchy which makes Julia's type system more than just a collection of object implementations. For example:

```
abstract type Number end
abstract type Real <: Number end
```

`Number` has no supertype, whereas `Real` is an abstract subtype of `Number`.

source

`primitive type` – Keyword.

```
primitive type
```

`primitive type` declares a concrete type whose data consists only of a series of bits. Classic examples of primitive types are integers and floating-point values. Some example built-in primitive type declarations:

```
primitive type Char 32 end
primitive type Bool <: Integer 8 end
```

The number after the name indicates how many bits of storage the type requires. Currently, only sizes that are multiples of 8 bits are supported. The `Bool` declaration shows how a primitive type can be optionally declared to be a subtype of some supertype.

source

`where` – Keyword.

```
where
```

The `where` keyword creates a `UnionAll` type, which may be thought of as an iterated union of other types, over all values of some variable. For example `Vector{T} where T<:Real` includes all `Vector`s where the element type is some kind of `Real` number.

The variable bound defaults to `Any` if it is omitted:

```
Vector{T} where T    # short for `where T<:Any`
```

Variables can also have lower bounds:

```
Vector{T} where T>:Int
Vector{T} where Int<:T<:Real
```

There is also a concise syntax for nested `where` expressions. For example, this:

```
Pair{T, S} where S<:Array{T} where T<:Number
```

can be shortened to:

```
Pair{T, S} where {T<:Number, S<:Array{T}}
```

This form is often found on method signatures.

Note that in this form, the variables are listed outermost-first. This matches the order in which variables are substituted when a type is "applied" to parameter values using the syntax `T{p1, p2, ...}`.

source

`...` – Keyword.

```
...
```

The "splat" operator, `...`, represents a sequence of arguments. `...` can be used in function definitions, to indicate that the function accepts an arbitrary number of arguments. `...` can also be used to apply a function to a sequence of arguments.

Examples

```
julia> add(xs...) = reduce(+, xs)
add (generic function with 1 method)

julia> add(1, 2, 3, 4, 5)
15

julia> add([1, 2, 3]...)
6

julia> add(7, 1:100..., 1000:1100...)
111107
```

source

`;` – Keyword.

```
;
```

Semicolons are used as statement separators and mark the beginning of keyword arguments in function declarations or calls.

source

= – Keyword.

```
=
```

= is the assignment operator.

- For variable a and expression b, `a = b` makes a refer to the value of b.
- For functions `f(x)`, `f(x) = x` defines a new function constant `f`, or adds a new method to `f` if `f` is already defined; this usage is equivalent to `function f(x); x; end`.
- `a[i] = v` calls `setindex!(a,v,i)`.
- `a.b = c` calls `setproperty!(a,:b,c)`.
- Inside a function call, `f(a=b)` passes b as the value of keyword argument a.
- Inside parentheses with commas, `(a=1,)` constructs a `NamedTuple`.

Examples

Assigning a to b does not create a copy of b; instead use `copy` or `deepcopy`.

```
julia> b = [1]; a = b; b[1] = 2; a
1-element Vector{Int64}:
 2

julia> b = [1]; a = copy(b); b[1] = 2; a
1-element Vector{Int64}:
 1
```

Collections passed to functions are also not copied. Functions can modify (mutate) the contents of the objects their arguments refer to. (The names of functions which do this are conventionally suffixed with '!'.)

```
julia> function f!(x); x[:] .+= 1; end
f! (generic function with 1 method)

julia> a = [1]; f!(a); a
1-element Vector{Int64}:
 2
```

Assignment can operate on multiple variables in parallel, taking values from an iterable:

```
julia> a, b = 4, 5
(4, 5)
```

```
julia> a, b = 1:3
1:3

julia> a, b
(1, 2)
```

Assignment can operate on multiple variables in series, and will return the value of the right-hand-most expression:

```
julia> a = [1]; b = [2]; c = [3]; a = b = c
1-element Vector{Int64}:
 3

julia> b[1] = 2; a, b, c
([2], [2], [2])
```

Assignment at out-of-bounds indices does not grow a collection. If the collection is a `Vector` it can instead be grown with `push!` or `append!`.

```
julia> a = [1, 1]; a[3] = 2
ERROR: BoundsError: attempt to access 2-element Vector{Int64} at index [3]
[...]

julia> push!(a, 2, 3)
4-element Vector{Int64}:
 1
 1
 2
 3
```

Assigning `[]` does not eliminate elements from a collection; instead use `filter!`.

```
julia> a = collect(1:3); a[a .<= 1] = []
ERROR: DimensionMismatch: tried to assign 0 elements to 1 destinations
[...]

julia> filter!(x -> x > 1, a) # in-place & thus more efficient than a = a[a .> 1]
2-element Vector{Int64}:
 2
 3
```

source

`?:` – Keyword.

```
a ? b : c
```

Short form for conditionals; read "if a, evaluate b otherwise evaluate c". Also known as the ternary operator.

This syntax is equivalent to `if a; b else c end`, but is often used to emphasize the value b-or-c which is being used as part of a larger expression, rather than the side effects that evaluating b or c may have.

See the manual section on control flow for more details.

Examples

```
julia> x = 1; y = 2;

julia> x > y ? println("x is larger") : println("x is not larger")
x is not larger

julia> x > y ? "x is larger" : x == y ? "x and y are equal" : "y is larger"
"y is larger"
```

source

`.=` – Keyword.

```
.=
```

Perform broadcasted assignment. The right-side argument is expanded as in `broadcast` and then assigned into the left-side argument in-place. Fuses with other dotted operators in the same expression; i.e. the whole assignment expression is converted into a single loop.

`A .= B` is similar to `broadcast!(identity, A, B)`.

Examples

```
julia> A = zeros(4, 4); B = [1, 2, 3, 4];

julia> A .= B
4×4 Matrix{Float64}:
 1.0  1.0  1.0  1.0
 2.0  2.0  2.0  2.0
 3.0  3.0  3.0  3.0
 4.0  4.0  4.0  4.0

julia> A
4×4 Matrix{Float64}:
 1.0  1.0  1.0  1.0
 2.0  2.0  2.0  2.0
 3.0  3.0  3.0  3.0
 4.0  4.0  4.0  4.0
```

source

`.` – Keyword.

```
.
```

The dot operator is used to access fields or properties of objects and access variables defined inside modules.

In general, `a.b` calls `getproperty(a, :b)` (see `getproperty`).

Examples

```
julia> z = 1 + 2im; z.im
2

julia> Iterators.product
product (generic function with 1 method)
```

source

`->` – Keyword.

```
x -> y
```

Create an anonymous function mapping argument(s) `x` to the function body `y`.

```
julia> f = x -> x^2 + 2x - 1
#1 (generic function with 1 method)

julia> f(2)
7
```

Anonymous functions can also be defined for multiple arguments.

```
julia> g = (x,y) -> x^2 + y^2
#2 (generic function with 1 method)

julia> g(2,3)
13
```

See the manual section on anonymous functions for more details.

source

`Base.:(:)` – Function.

```
:expr
```

Quote an expression `expr`, returning the abstract syntax tree (AST) of `expr`. The AST may be of type `Expr`, `Symbol`, or a literal value. The syntax `:identifier` evaluates to a `Symbol`.

See also: `Expr`, `Symbol`, `Meta.parse`

Examples

```
julia> expr = :(a = b + 2*x)
:(a = b + 2x)

julia> sym = :some_identifier
:some_identifier
```

```
julia> value = :0xff
0xff

julia> typeof((expr, sym, value))
Tuple{Expr, Symbol, UInt8}
```

source

`::` – Keyword.

```
::
```

The `::` operator either asserts that a value has the given type, or declares that a local variable or function return always has the given type.

Given `expression::T`, `expression` is first evaluated. If the result is of type `T`, the value is simply returned. Otherwise, a `TypeError` is thrown.

In local scope, the syntax `local x::T` or `x::T = expression` declares that local variable `x` always has type `T`. When a value is assigned to the variable, it will be converted to type `T` by calling `convert`.

In a method declaration, the syntax `function f(x)::T` causes any value returned by the method to be converted to type `T`.

See the manual section on Type Declarations.

Examples

```
julia> (1+2)::AbstractFloat
ERROR: TypeError: typeassert: expected AbstractFloat, got a value of type Int64

julia> (1+2)::Int
3

julia> let
           local x::Int
           x = 2.0
           x
       end
2
```

source

`[]` – Keyword.

```
[]
```

Square brackets are used for indexing (`getindex`), indexed assignment (`setindex!`), array literals (`Base.vect`), array concatenation (`vcat`, `hcat`, `hvcat`, `hvncat`), and array comprehensions (`collect`).

source

44.4 Standard Modules

`Main` – Module.

```
Main
```

`Main` is the top-level module, and Julia starts with `Main` set as the current module. Variables defined at the prompt go in `Main`, and `varinfo` lists variables in `Main`.

```
julia> @__MODULE__
Main
```

source

`Core` – Module.

```
Core
```

`Core` is the module that contains all identifiers considered "built in" to the language, i.e. part of the core language and not libraries. Every module implicitly specifies `using Core`, since you can't do anything without those definitions.

source

`Base` – Module.

```
Base
```

The base library of Julia. `Base` is a module that contains basic functionality (the contents of `base/`). All modules implicitly contain `using Base`, since this is needed in the vast majority of cases.

source

44.5 Base Submodules

`Base.Broadcast` – Module.

```
Base.Broadcast
```

Module containing the broadcasting implementation.

source

`Base.Docs` – Module.

```
Docs
```

The `Docs` module provides the `@doc` macro which can be used to set and retrieve documentation metadata for Julia objects.

Please see the manual section on documentation for more information.

source

`Base.Iterators` – Module.

Methods for working with Iterators.

source

`Base.Libc` – Module.

Interface to libc, the C standard library.

source

`Base.Meta` – Module.

Convenience functions for metaprogramming.

source

`Base.StackTraces` – Module.

Tools for collecting and manipulating stack traces. Mainly used for building errors.

source

`Base.Sys` – Module.

Provide methods for retrieving information about hardware and the operating system.

source

`Base.Threads` – Module.

Multithreading support.

source

`Base.GC` – Module.

```
Base.GC
```

Module with garbage collection utilities.

source

44.6 All Objects

`Core.:(===)` – Function.

```
===(x,y)::Bool
≡(x,y)::Bool
```

Determine whether x and y are identical, in the sense that no program could distinguish them. First the types of x and y are compared. If those are identical, mutable objects are compared by address in memory and immutable objects (such as numbers) are compared by contents at the bit level. This function is sometimes called "egal". It always returns a `Bool` value.

Examples

```
julia> a = [1, 2]; b = [1, 2];

julia> a == b
true

julia> a === b
false

julia> a === a
true
```

source

`Core.isa` – Function.

```
isa(x, type)::Bool
```

Determine whether `x` is of the given `type`. Can also be used as an infix operator, e.g. `x isa type`.

Examples

```
julia> isa(1, Int)
true

julia> isa(1, Matrix)
false

julia> isa(1, Char)
false

julia> isa(1, Number)
true

julia> 1 isa Number
true
```

source

`Base.isequal` – Function.

```
isequal(x)
```

Create a function that compares its argument to `x` using `isequal`, i.e. a function equivalent to `y -> isequal(y, x)`.

The returned function is of type `Base.Fix2{typeof(isequal)}`, which can be used to implement specialized methods.

source

```
isequal(x, y)::Bool
```

Similar to `==`, except for the treatment of floating point numbers and of missing values. `isequal` treats all floating-point `NaN` values as equal to each other, treats `-0.0` as unequal to `0.0`, and `missing` as equal to `missing`. Always returns a `Bool` value.

`isequal` is an equivalence relation - it is reflexive (`===` implies `isequal`), symmetric (`isequal(a, b)` implies `isequal(b, a)`) and transitive (`isequal(a, b)` and `isequal(b, c)` implies `isequal(a, c)`).

Implementation

The default implementation of `isequal` calls `==`, so a type that does not involve floating-point values generally only needs to define `==`.

`isequal` is the comparison function used by hash tables (`Dict`). `isequal(x,y)` must imply that `hash(x) == hash(y)`.

This typically means that types for which a custom `==` or `isequal` method exists must implement a corresponding `hash` method (and vice versa). Collections typically implement `isequal` by calling `isequal` recursively on all contents.

Furthermore, `isequal` is linked with `isless`, and they work together to define a fixed total ordering, where exactly one of `isequal(x, y)`, `isless(x, y)`, or `isless(y, x)` must be `true` (and the other two `false`).

Scalar types generally do not need to implement `isequal` separate from `==`, unless they represent floating-point numbers amenable to a more efficient implementation than that provided as a generic fallback (based on `isnan`, `signbit`, and `==`).

Examples

```
julia> isequal([1., NaN], [1., NaN])
true

julia> [1., NaN] == [1., NaN]
false

julia> 0.0 == -0.0
true

julia> isequal(0.0, -0.0)
false

julia> missing == missing
missing

julia> isequal(missing, missing)
true
```

source

Base.isless – Function.

```
isless(x, y)
```

Test whether x is less than y, according to a fixed total order (defined together with `isequal`). `isless` is not defined for pairs `(x, y)` of all types. However, if it is defined, it is expected to satisfy the following:

- If `isless(x, y)` is defined, then so is `isless(y, x)` and `isequal(x, y)`, and exactly one of those three yields `true`.

- The relation defined by `isless` is transitive, i.e., `isless(x, y) && isless(y, z)` implies `isless(x, z)`.

Values that are normally unordered, such as `NaN`, are ordered after regular values. `missing` values are ordered last.

This is the default comparison used by `sort!`.

Implementation

Non-numeric types with a total order should implement this function. Numeric types only need to implement it if they have special values such as `NaN`. Types with a partial order should implement `<`. See the documentation on Alternate Orderings for how to define alternate ordering methods that can be used in sorting and related functions.

Examples

```
julia> isless(1, 3)
true

julia> isless("Red", "Blue")
false
```

source

`Base.ispositive` – Function.

```
ispositive(x)
```

Test whether `x > 0`. See also `isnegative`.

Julia 1.13

This function requires at least Julia 1.13.

Examples

```
julia> ispositive(-4.0)
false

julia> ispositive(99)
true

julia> ispositive(0.0)
false
```

source

`Base.isnegative` – Function.

```
isnegative(x)
```

Test whether `x < 0`. See also `ispositive`.

Julia 1.13

This function requires at least Julia 1.13.

Examples

```
julia> isnegative(-4.0)
true

julia> isnegative(99)
false

julia> isnegative(-0.0)
false
```

source

`Base.isunordered` – Function.

```
isunordered(x)
```

Return `true` if `x` is a value that is not orderable according to `<`, such as `NaN` or `missing`.

The values that evaluate to `true` with this predicate may be orderable with respect to other orderings such as `isless`.

Julia 1.7

This function requires Julia 1.7 or later.

source

`Base.ifelse` – Function.

```
ifelse(condition::Bool, x, y)
```

Return `x` if `condition` is `true`, otherwise return `y`. This differs from `?` or `if` in that it is an ordinary function, so all the arguments are evaluated first. In some cases, using `ifelse` instead of an `if` statement can eliminate the branch in generated code and provide higher performance in tight loops.

Examples

```
julia> ifelse(1 > 2, 1, 2)
2
```

source

`Core.typeassert` – Function.

```
typeassert(x, type)
```

Throw a `TypeError` unless `x isa type`. The syntax `x::type` calls this function.

Examples

```
julia> typeassert(2.5, Int)
ERROR: TypeError: in typeassert, expected Int64, got a value of type Float64
Stacktrace:
[...]
```

source

Core.typeof – Function.

```
typeof(x)
```

Get the concrete type of x.

See also eltype.

Examples

```
julia> a = 1//2;

julia> typeof(a)
Rational{Int64}

julia> M = [1 2; 3.5 4];

julia> typeof(M)
Matrix{Float64} (alias for Array{Float64, 2})
```

source

Core.tuple – Function.

```
tuple(xs...)
```

Construct a tuple of the given objects.

See also Tuple, ntuple, NamedTuple.

Examples

```
julia> tuple(1, 'b', pi)
(1, 'b', π)

julia> ans === (1, 'b', π)
true

julia> Tuple(Real[1, 2, pi])  # takes a collection
(1, 2, π)
```

source

Base.ntuple – Function.

```
ntuple(f, ::Val{N})
```

Create a tuple of length `N`, computing each element as `f(i)`, where `i` is the index of the element. By taking a `Val(N)` argument, it is possible that this version of ntuple may generate more efficient code than the version taking the length as an integer. But `ntuple(f, N)` is preferable to `ntuple(f, Val(N))` in cases where `N` cannot be determined at compile time.

Examples

```
julia> ntuple(i -> 2*i, Val(4))
(2, 4, 6, 8)
```

source

```
ntuple(f, n::Integer)
```

Create a tuple of length `n`, computing each element as `f(i)`, where `i` is the index of the element.

Examples

```
julia> ntuple(i -> 2*i, 4)
(2, 4, 6, 8)
```

source

`Base.objectid` – Function.

```
objectid(x)::UInt
```

Get a hash value for `x` based on object identity. This value is not unique nor stable between Julia processes or versions.

If `x === y` then `objectid(x) == objectid(y)`, and usually when `x !== y`, `objectid(x) != objectid(y)`.

See also `hash`, `IdDict`.

source

`Base.hash` – Function.

```
hash(x[, h::UInt])::UInt
```

Compute an integer hash code such that `isequal(x,y)` implies `isequal(hash(x), hash(y))`. The optional second argument `h` is another hash code to be mixed with the result.

New types should implement the 2-argument form, typically by calling the 2-argument hash method recursively in order to mix hashes of the contents with each other (and with `h`). Typically, any type that implements hash should also implement its own `==` (hence `isequal`) to guarantee the property mentioned above.

The hash value may change when a new Julia process is started.

Warning

When implementing the 2-argument form, the second argument `h` should *not* be given a default value such `h = UInt(0)` as this will implicitly create a 1-argument method that is more specific than the fallback (see Note on Optional and keyword Arguments), but potentially with the wrong seed, causing hash inconsistencies.

```
julia> a = hash(10)
0x759d18cc5346a65f

julia> hash(10, a) # only use the output of another hash function as the second argument
0x03158cd61b1b0bd1
```

See also: `objectid`, `Dict`, `Set`.

source

`Base.finalizer` – Function.

```
finalizer(f, x)
```

Register a function `f(x)` to be called when there are no program-accessible references to `x`, and return `x`. The type of `x` must be a `mutable struct`, otherwise the function will throw.

`f` must not cause a task switch, which excludes most I/O operations such as `println`. Using the `@async` macro (to defer context switching to outside of the finalizer) or `ccall` to directly invoke IO functions in C may be helpful for debugging purposes.

Note that there is no guaranteed world age for the execution of `f`. It may be called in the world age in which the finalizer was registered or any later world age.

Examples

```
finalizer(my_mutable_struct) do x
    @async println("Finalizing $x.")
end

finalizer(my_mutable_struct) do x
    ccall(:jl_safe_printf, Cvoid, (Cstring, Cstring), "Finalizing %s.", repr(x))
end
```

A finalizer may be registered at object construction. In the following example note that we implicitly rely on the finalizer returning the newly created mutable struct `x`.

```
mutable struct MyMutableStruct
    bar
    function MyMutableStruct(bar)
        x = new(bar)
        f(t) = @async println("Finalizing $t.")
        finalizer(f, x)
    end
end
```

source

`Base.finalize` – Function.

```
finalize(x)
```

Immediately run finalizers registered for object `x`.

source

Base.copy – Function.

```
copy(x)
```

Create a shallow copy of x: the outer structure is copied, but not all internal values. For example, copying an array produces a new array with identically-same elements as the original.

See also copy!, copyto!, deepcopy.

source

Base.deepcopy – Function.

```
deepcopy(x)
```

Create a deep copy of x: everything is copied recursively, resulting in a fully independent object. For example, deep-copying an array creates deep copies of all the objects it contains and produces a new array with the consistent relationship structure (e.g., if the first two elements are the same object in the original array, the first two elements of the new array will also be the same deepcopyed object). Calling deepcopy on an object should generally have the same effect as serializing and then deserializing it.

While it isn't normally necessary, user-defined types can override the default deepcopy behavior by defining a specialized version of the function deepcopy_internal(x::T, dict::IdDict) (which shouldn't otherwise be used), where T is the type to be specialized for, and dict keeps track of objects copied so far within the recursion. Within the definition, deepcopy_internal should be used in place of deepcopy, and the dict variable should be updated as appropriate before returning.

Warning

It is better to avoid this function in favor of custom copy methods or use-case-specific copying functions. deepcopy is slow and can easily copy too many objects, or generate an object that violates invariants, since it does not respect abstraction boundaries.

source

Base.getproperty – Function.

```
getproperty(value, name::Symbol)
getproperty(value, name::Symbol, order::Symbol)
```

The syntax a.b calls getproperty(a, :b). The syntax @atomic order a.b calls getproperty(a, :b, :order) and the syntax @atomic a.b calls getproperty(a, :b, :sequentially_consistent).

Examples

```
julia> struct MyType{T <: Number}
           x::T
       end

julia> function Base.getproperty(obj::MyType, sym::Symbol)
           if sym === :special
               return obj.x + 1
           else # fallback to getfield
               return getfield(obj, sym)
```

```
            end
        end

julia> obj = MyType(1);

julia> obj.special
2

julia> obj.x
1
```

One should overload `getproperty` only when necessary, as it can be confusing if the behavior of the syntax `obj.f` is unusual. Also note that using methods is often preferable. See also this style guide documentation for more information: Prefer exported methods over direct field access.

See also `getfield`, `propertynames` and `setproperty!`.

source

`Base.setproperty!` – Function.

```
setproperty!(value, name::Symbol, x)
setproperty!(value, name::Symbol, x, order::Symbol)
```

The syntax `a.b = c` calls `setproperty!(a, :b, c)`. The syntax `@atomic order a.b = c` calls `setproperty!(a, :b, c, :order)` and the syntax `@atomic a.b = c` calls `setproperty!(a, :b, c, :sequentially_consistent)`.

Julia 1.8

`setproperty!` on modules requires at least Julia 1.8.

See also `setfield!`, `propertynames` and `getproperty`.

source

`Base.replaceproperty!` – Function.

```
replaceproperty!(x, f::Symbol, expected, desired, success_order::Symbol=:not_atomic,
↪  fail_order::Symbol=success_order)
```

Perform a compare-and-swap operation on `x.f` from `expected` to `desired`, per egal. The syntax `@atomicreplace x.f expected => desired` can be used instead of the function call form.

See also `replacefield! setproperty!`, `setpropertyonce!`.

source

`Base.swapproperty!` – Function.

```
swapproperty!(x, f::Symbol, v, order::Symbol=:not_atomic)
```

The syntax `@atomic a.b, _ = c, a.b` returns `(c, swapproperty!(a, :b, c, :sequentially_consistent))`, where there must be one `getproperty` expression common to both sides.

See also `swapfield!` and `setproperty!`.

source

Base.modifyproperty! – Function.

```
modifyproperty!(x, f::Symbol, op, v, order::Symbol=:not_atomic)
```

The syntax `@atomic op(x.f, v)` (and its equivalent `@atomic x.f op v`) returns `modifyproperty!(x, :f, op, v, :sequentially_consistent)`, where the first argument must be a `getproperty` expression and is modified atomically.

Invocation of `op(getproperty(x, f), v)` must return a value that can be stored in the field `f` of the object `x` by default. In particular, unlike the default behavior of `setproperty!`, the `convert` function is not called automatically.

See also `modifyfield!` and `setproperty!`.

source

Base.setpropertyonce! – Function.

```
setpropertyonce!(x, f::Symbol, value, success_order::Symbol=:not_atomic,
↪  fail_order::Symbol=success_order)
```

Perform a compare-and-swap operation on `x.f` to set it to `value` if previously unset. The syntax `@atomiconce x.f = value` can be used instead of the function call form.

See also `setfieldonce!`, `setproperty!`, `replaceproperty!`.

Julia 1.11

This function requires Julia 1.11 or later.

source

Base.propertynames – Function.

```
propertynames(x, private=false)
```

Get a tuple or a vector of the properties (`x.property`) of an object `x`. This is typically the same as `fieldnames(typeof(x))`, but types that overload `getproperty` should generally overload `propertynames` as well to get the properties of an instance of the type.

`propertynames(x)` may return only "public" property names that are part of the documented interface of `x`. If you want it to also return "private" property names intended for internal use, pass `true` for the optional second argument. REPL tab completion on `x.` shows only the `private=false` properties.

See also: `hasproperty`, `hasfield`.

source

Base.hasproperty – Function.

```
hasproperty(x, s::Symbol)
```

Return a boolean indicating whether the object `x` has `s` as one of its own properties.

Julia 1.2

This function requires at least Julia 1.2.

See also: propertynames, hasfield.

source

Core.getfield – Function.

```
getfield(value, name::Symbol, [order::Symbol], [boundscheck::Bool=true])
getfield(value, i::Int, [order::Symbol], [boundscheck::Bool=true])
```

Extract a field from a composite value by name or position.

Optionally, an ordering can be defined for the operation. If the field was declared @atomic, the specification is strongly recommended to be compatible with the stores to that location. Otherwise, if not declared as @atomic, this parameter must be :not_atomic if specified.

The bounds check may be disabled, in which case the behavior of this function is undefined if i is out of bounds.

See also getproperty and fieldnames.

Examples

```
julia> a = 1//2
1//2

julia> getfield(a, :num)
1

julia> a.num
1

julia> getfield(a, 1)
1
```

source

Core.setfield! – Function.

```
setfield!(value, name::Symbol, x, [order::Symbol])
setfield!(value, i::Int, x, [order::Symbol])
```

Assign x to a named field in value of composite type. The value must be mutable and x must be a subtype of fieldtype(typeof(value), name). Additionally, an ordering can be specified for this operation. If the field was declared @atomic, this specification is mandatory. Otherwise, if not declared as @atomic, it must be :not_atomic if specified. See also setproperty!.

Examples

```
julia> mutable struct MyMutableStruct
           field::Int
       end

julia> a = MyMutableStruct(1);

julia> setfield!(a, :field, 2);
```

```
julia> getfield(a, :field)
2

julia> a = 1//2
1//2

julia> setfield!(a, :num, 3);
ERROR: setfield!: immutable struct of type Rational cannot be changed
```

source

`Core.modifyfield!` – Function.

```
modifyfield!(value, name::Symbol, op, x, [order::Symbol])::Pair
modifyfield!(value, i::Int, op, x, [order::Symbol])::Pair
```

Atomically perform the operations to get and set a field after applying the function op.

```
y = getfield(value, name)
z = op(y, x)
setfield!(value, name, z)
return y => z
```

If supported by the hardware (for example, atomic increment), this may be optimized to the appropriate hardware instruction, otherwise it'll use a loop.

Julia 1.7

This function requires Julia 1.7 or later.

source

`Core.replacefield!` – Function.

```
replacefield!(value, name::Symbol, expected, desired,
              [success_order::Symbol, [fail_order::Symbol=success_order]) -> (; old,
              ↪  success::Bool)
replacefield!(value, i::Int, expected, desired,
              [success_order::Symbol, [fail_order::Symbol=success_order]) -> (; old,
              ↪  success::Bool)
```

Atomically perform the operations to get and conditionally set a field to a given value.

```
y = getfield(value, name, fail_order)
ok = y === expected
if ok
    setfield!(value, name, desired, success_order)
end
return (; old = y, success = ok)
```

If supported by the hardware, this may be optimized to the appropriate hardware instruction, otherwise it'll use a loop.

Julia 1.7

This function requires Julia 1.7 or later.

source

`Core.swapfield!` – Function.

```
swapfield!(value, name::Symbol, x, [order::Symbol])
swapfield!(value, i::Int, x, [order::Symbol])
```

Atomically perform the operations to simultaneously get and set a field:

```
y = getfield(value, name)
setfield!(value, name, x)
return y
```

Julia 1.7

This function requires Julia 1.7 or later.

source

`Core.setfieldonce!` – Function.

```
setfieldonce!(value, name::Union{Int,Symbol}, desired,
              [success_order::Symbol, [fail_order::Symbol=success_order]) -> success::Bool
```

Atomically perform the operations to set a field to a given value, only if it was previously not set.

```
ok = !isdefined(value, name, fail_order)
if ok
    setfield!(value, name, desired, success_order)
end
return ok
```

Julia 1.11

This function requires Julia 1.11 or later.

source

`Core.isdefined` – Function.

```
isdefined(m::Module, s::Symbol, [order::Symbol])
isdefined(object, s::Symbol, [order::Symbol])
isdefined(object, index::Int, [order::Symbol])
```

Tests whether a global variable or object field is defined. The arguments can be a module and a symbol or a composite object and field name (as a symbol) or index. Optionally, an ordering can be defined for the operation. If the field was declared `@atomic`, the specification is strongly recommended to be compatible with the stores to that location. Otherwise, if not declared as `@atomic`, this parameter must be `:not_atomic` if specified.

To test whether an array element is defined, use `isassigned` instead.

The global variable variant is supported for compatibility with older julia releases. For new code, prefer `isdefinedglobal`.

See also `@isdefined`.

Examples

```
julia> isdefined(Base, :sum)
true

julia> isdefined(Base, :NonExistentMethod)
false

julia> a = 1//2;

julia> isdefined(a, 2)
true

julia> isdefined(a, 3)
false

julia> isdefined(a, :num)
true

julia> isdefined(a, :numerator)
false
```

source

`Core.isdefinedglobal` – Function.

```
isdefinedglobal(m::Module, s::Symbol, [allow_import::Bool=true, [order::Symbol=:unordered]])
```

Tests whether a global variable `s` is defined in module `m` (in the current world age). A variable is considered defined if and only if a value may be read from this global variable and an access will not throw. This includes both constants and global variables that have a value set.

If `allow_import` is `false`, the global variable must be defined inside `m` and may not be imported from another module.

Julia 1.12

This function requires Julia 1.12 or later.

See also `@isdefined`.

Examples

```
julia> isdefinedglobal(Base, :sum)
true

julia> isdefinedglobal(Base, :NonExistentMethod)
false
```

```
julia> isdefinedglobal(Base, :sum, false)
true

julia> isdefinedglobal(Main, :sum, false)
false
```

source

Base.@isdefined – Macro.

```
@isdefined(s)::Bool
```

Tests whether variable s is defined in the current scope.

See also isdefined for field properties and isassigned for array indexes or haskey for other mappings.

Examples

```
julia> @isdefined newvar
false

julia> newvar = 1
1

julia> @isdefined newvar
true

julia> function f()
           println(@isdefined x)
           x = 3
           println(@isdefined x)
       end
f (generic function with 1 method)

julia> f()
false
true
```

source

Base.convert – Function.

```
convert(T, x)
```

Convert x to a value of type T.

If T is an Integer type, an InexactError will be raised if x is not representable by T, for example if x is not integer-valued, or is outside the range supported by T.

Examples

```
julia> convert(Int, 3.0)
3
```

```
julia> convert(Int, 3.5)
ERROR: InexactError: Int64(3.5)
Stacktrace:
[...]
```

If T is an `AbstractFloat` type, then it will return the closest value to x representable by T. Inf is treated as one ulp greater than `floatmax(T)` for purposes of determining nearest.

```
julia> x = 1/3
0.3333333333333333

julia> convert(Float32, x)
0.33333334f0

julia> convert(BigFloat, x)
0.333333333333333314829616256247390992939472198486328125
```

If T is a collection type and x a collection, the result of `convert(T, x)` may share memory with all or part of x.

```
julia> x = Int[1, 2, 3];

julia> y = convert(Vector{Int}, x);

julia> y === x
true
```

See also: `round`, `trunc`, `oftype`, `reinterpret`.

source

`Base.promote` – Function.

```
promote(xs...)
```

Convert all arguments to a common type, and return them all (as a tuple). If no arguments can be converted, an error is raised.

See also: `promote_type`, `promote_rule`.

Examples

```
julia> promote(Int8(1), Float16(4.5), Float32(4.1))
(1.0f0, 4.5f0, 4.1f0)

julia> promote_type(Int8, Float16, Float32)
Float32

julia> reduce(Base.promote_typejoin, (Int8, Float16, Float32))
Real

julia> promote(1, "x")
ERROR: promotion of types Int64 and String failed to change any arguments
[...]
```

```
julia> promote_type(Int, String)
Any
```

source

Base.oftype – Function.

```
oftype(x, y)
```

Convert y to the type of x i.e. `convert(typeof(x), y)`.

Examples

```
julia> x = 4;

julia> y = 3.;

julia> oftype(x, y)
3

julia> oftype(y, x)
4.0
```

source

Base.widen – Function.

```
widen(x)
```

If x is a type, return a "larger" type, defined so that arithmetic operations + and - are guaranteed not to overflow nor lose precision for any combination of values that type x can hold.

For fixed-size integer types less than 128 bits, `widen` will return a type with twice the number of bits.

If x is a value, it is converted to `widen(typeof(x))`.

Examples

```
julia> widen(Int32)
Int64

julia> widen(1.5f0)
1.5
```

source

Base.identity – Function.

```
identity(x)
```

The identity function. Returns its argument.

See also: `one`, `oneunit`, and `LinearAlgebra`'s `I`.

Examples

```
julia> identity("Well, what did you expect?")
"Well, what did you expect?"
```

source

Core.WeakRef – Type.

```
WeakRef(x)
```

w = WeakRef(x) constructs a weak reference to the Julia value x: although w contains a reference to x, it does not prevent x from being garbage collected. w.value is either x (if x has not been garbage-collected yet) or nothing (if x has been garbage-collected).

```
julia> x = "a string"
"a string"

julia> w = WeakRef(x)
WeakRef("a string")

julia> GC.gc()

julia> w           # a reference is maintained via `x`
WeakRef("a string")

julia> x = nothing # clear reference

julia> GC.gc()

julia> w
WeakRef(nothing)
```

source

44.7 Properties of Types

Type relations

Base.supertype – Function.

```
supertype(T::Union{DataType, UnionAll})
```

Return the direct supertype of type T. T can be a DataType or a UnionAll type. Does not support type Unions. Also see info on Types.

Examples

```
julia> supertype(Int32)
Signed

julia> supertype(Vector)
DenseVector (alias for DenseArray{T, 1} where T)
```

source

`Core.Type` – Type.

```
Core.Type{T}
```

`Core.Type` is an abstract type which has all type objects as its instances. The only instance of the singleton type `Core.Type{T}` is the object `T`.

Examples

```
julia> isa(Type{Float64}, Type)
true

julia> isa(Float64, Type)
true

julia> isa(Real, Type{Float64})
false

julia> isa(Real, Type{Real})
true
```

source

`Core.DataType` – Type.

```
DataType <: Type{T}
```

`DataType` represents explicitly declared types that have names, explicitly declared supertypes, and, optionally, parameters. Every concrete value in the system is an instance of some `DataType`.

Examples

```
julia> typeof(Real)
DataType

julia> typeof(Int)
DataType

julia> struct Point
           x::Int
           y
       end

julia> typeof(Point)
DataType
```

source

`Core.:<:` – Function.

```
<:(T1, T2)::Bool
```

Subtyping relation, defined between two types. In Julia, a type `S` is said to be a *subtype* of a type `T` if and only if we have `S <: T`.

For any type L and any type R, `L <: R` implies that any value v of type L is also of type R. I.e., `(L <: R) && (v isa L)` implies `v isa R`.

The subtyping relation is a *partial order*. I.e., `<:` is:

- *reflexive*: for any type T, `T <: T` holds
- *antisymmetric*: for any type A and any type B, `(A <: B) && (B <: A)` implies `A == B`
- *transitive*: for any type A, any type B and any type C; `(A <: B) && (B <: C)` implies `A <: C`

See also info on Types, `Union{}`, `Any`, `isa`.

Examples

```
julia> Float64 <: AbstractFloat
true

julia> Vector{Int} <: AbstractArray
true

julia> Matrix{Float64} <: Matrix{AbstractFloat}  # `Matrix` is invariant
false

julia> Tuple{Float64} <: Tuple{AbstractFloat}    # `Tuple` is covariant
true

julia> Union{} <: Int  # The bottom type, `Union{}`, subtypes each type.
true

julia> Union{} <: Float32 <: AbstractFloat <: Real <: Number <: Any  # Operator chaining
true
```

The `<:` keyword also has several syntactic uses which represent the same subtyping relation, but which do not execute the operator or return a Bool:

- To specify the lower bound and the upper bound on a parameter of a `UnionAll` type in a `where` statement.
- To specify the lower bound and the upper bound on a (static) parameter of a method, see Parametric Methods.
- To define a subtyping relation while declaring a new type, see `struct` and `abstract type`.

source

`Core.:>:` – Function.

```
>:(T1, T2)
```

Supertype operator, equivalent to `T2 <: T1`.

source

`Base.typejoin` – Function.

```
typejoin(T, S, ...)
```

Return the closest common ancestor of types T and S, i.e. the narrowest type from which they both inherit. Recurses on additional varargs.

Examples

```
julia> typejoin(Int, Float64)
Real

julia> typejoin(Int, Float64, ComplexF32)
Number
```

source

Base.typeintersect – Function.

```
typeintersect(T::Type, S::Type)
```

Compute a type that contains the intersection of T and S. Usually this will be the smallest such type or one close to it.

A special case where exact behavior is guaranteed: when `T <: S`, `typeintersect(S, T) == T == typeintersect(T, S)`.

source

Base.promote_type – Function.

```
promote_type(type1, type2, ...)
```

Promotion refers to converting values of mixed types to a single common type. `promote_type` represents the default promotion behavior in Julia when operators (usually mathematical) are given arguments of differing types. `promote_type` generally tries to return a type which can at least approximate most values of either input type without excessively widening. Some loss is tolerated; for example, `promote_type(Int64, Float64)` returns `Float64` even though strictly, not all `Int64` values can be represented exactly as `Float64` values.

See also: `promote`, `promote_typejoin`, `promote_rule`.

Examples

```
julia> promote_type(Int64, Float64)
Float64

julia> promote_type(Int32, Int64)
Int64

julia> promote_type(Float32, BigInt)
BigFloat

julia> promote_type(Int16, Float16)
Float16

julia> promote_type(Int64, Float16)
```

```
Float16

julia> promote_type(Int8, UInt16)
UInt16
```

Don't overload this directly

To overload promotion for your own types you should overload `promote_rule`. `promote_type` calls `promote_rule` internally to determine the type. Overloading `promote_type` directly can cause ambiguity errors.

source

`Base.promote_rule` – Function.

```
promote_rule(type1, type2)
```

Specifies what type should be used by `promote` when given values of types `type1` and `type2`. This function should not be called directly, but should have definitions added to it for new types as appropriate.

source

`Base.promote_typejoin` – Function.

```
promote_typejoin(T, S)
```

Compute a type that contains both T and S, which could be either a parent of both types, or a `Union` if appropriate. Falls back to `typejoin`.

See instead `promote`, `promote_type`.

Examples

```
julia> Base.promote_typejoin(Int, Float64)
Real

julia> Base.promote_type(Int, Float64)
Float64
```

source

`Base.iskindtype` – Function.

```
Base.iskindtype(T)
```

Determine whether T is a kind, that is, the type of a Julia type: a `DataType`, `Union`, `UnionAll`, or `Core.TypeofBottom`.

All kinds are concrete because types are Julia values.

source

`Base.isdispatchtuple` – Function.

```
isdispatchtuple(T)
```

Determine whether type T is a `Tuple` that could appear as a type signature in dispatch. For this to be true, every element of the tuple type must be either:

- concrete but not a kind type
- a `Type{U}` with no free type variables in U

Note

A dispatch tuple is relevant for method dispatch because it has no inhabited subtypes.

For example, `Tuple{Int, DataType}` is concrete, but is not a dispatch tuple because `Tuple{Int, Type{Bool}}` is an inhabited subtype.

`Tuple{Tuple{DataType}}` *is* a dispatch tuple because `Tuple{DataType}` is concrete and not a kind; the subtype `Tuple{Tuple{Type{Int}}}` is not inhabited.

If T is not a type, then return `false`.

Examples

```
julia> isdispatchtuple(Int)
false

julia> isdispatchtuple(Tuple{Int})
true

julia> isdispatchtuple(Tuple{Number})
false

julia> isdispatchtuple(Tuple{DataType})
false

julia> isdispatchtuple(Tuple{Type{Int}})
true

julia> isdispatchtuple(Tuple{Type})
false
```

source

Declared structure

`Base.ismutable` – Function.

```
ismutable(v)::Bool
```

Return `true` if and only if value `v` is mutable. See Mutable Composite Types for a discussion of immutability. Note that this function works on values, so if you give it a `DataType`, it will tell you that a value of the type is mutable.

Note

For technical reasons, `ismutable` returns `true` for values of certain special types (for example `String` and `Symbol`) even though they cannot be mutated in a permissible way.

See also `isbits`, `isstructtype`.

Examples

```
julia> ismutable(1)
false

julia> ismutable([1,2])
true
```

Julia 1.5

This function requires at least Julia 1.5.

source

`Base.isimmutable` – Function.

```
isimmutable(v) -> Bool
```

Warning

Consider using `!ismutable(v)` instead, as `isimmutable(v)` will be replaced by `!ismutable(v)` in a future release. (Since Julia 1.5)

Return `true` iff value `v` is immutable. See Mutable Composite Types for a discussion of immutability. Note that this function works on values, so if you give it a type, it will tell you that a value of `DataType` is mutable.

Examples

```
julia> isimmutable(1)
true

julia> isimmutable([1,2])
false
```

source

`Base.ismutabletype` – Function.

```
ismutabletype(T)::Bool
```

Determine whether type T was declared as a mutable type (i.e. using `mutable struct` keyword). If T is not a type, then return `false`.

Julia 1.7

This function requires at least Julia 1.7.

source

Base.isabstracttype – Function.

```
isabstracttype(T)
```

Determine whether type T was declared as an abstract type (i.e. using the `abstract type` syntax). If T is not a type, then return `false`.

Note

While abstract types are not concrete and vice versa, types can be neither concrete nor abstract (for example, `Vector` (a `UnionAll`)).

See also: `isconcretetype`.

Examples

```
julia> isabstracttype(AbstractArray)
true

julia> isabstracttype(Vector)
false
```

source

Base.isprimitivetype – Function.

```
isprimitivetype(T)::Bool
```

Determine whether type T was declared as a primitive type (i.e. using the `primitive type` syntax). If T is not a type, then return `false`.

source

Base.issingletontype – Function.

```
Base.issingletontype(T)
```

Determine whether type T has exactly one possible instance; for example, a struct type with no fields except other singleton values. If T is not a concrete type, then return `false`.

source

Base.isstructtype – Function.

```
isstructtype(T)::Bool
```

Determine whether type T was declared as a struct type (i.e. using the `struct` or `mutable struct` keyword). If T is not a type, then return `false`.

source

Base.nameof – Method.

```
nameof(t::DataType)::Symbol
```

Get the name of a (potentially `UnionAll`-wrapped) `DataType` (without its parent module) as a symbol.

Examples

```
julia> module Foo
           struct S{T}
           end
       end
Foo

julia> nameof(Foo.S{T} where T)
:S
```

source

`Base.fieldnames` – Function.

```
fieldnames(x::DataType)
```

Get a tuple with the names of the fields of a `DataType`.

Each name is a `Symbol`, except when `x <: Tuple`, in which case each name (actually the index of the field) is an `Int`.

See also `propertynames`, `hasfield`.

Examples

```
julia> fieldnames(Rational)
(:num, :den)

julia> fieldnames(typeof(1+im))
(:re, :im)

julia> fieldnames(Tuple{String,Int})
(1, 2)
```

source

`Base.fieldname` – Function.

```
fieldname(x::DataType, i::Integer)
```

Get the name of field `i` of a `DataType`.

The return type is `Symbol`, except when `x <: Tuple`, in which case the index of the field is returned, of type `Int`.

Examples

```
julia> fieldname(Rational, 1)
:num

julia> fieldname(Rational, 2)
:den

julia> fieldname(Tuple{String,Int}, 2)
2
```

source

Base.fieldindex – Function.

```
fieldindex(T, name::Symbol, err:Bool=true)
```

Get the index of a named field, throwing an error if the field does not exist (when err==true) or returning 0 (when err==false).

Examples

```
julia> struct Foo
           x::Int64
           y::String
       end

julia> fieldindex(Foo, :y)
2

julia> fieldindex(Foo, :z)
ERROR: FieldError: type Foo has no field `z`, available fields: `x`, `y`
Stacktrace:
[...]

julia> fieldindex(Foo, :z, false)
0
```

Julia 1.13

This function is exported as of Julia 1.13.

source

Core.fieldtype – Function.

```
fieldtype(T, name::Symbol | index::Int)
```

Determine the declared type of a field (specified by name or index) in a composite DataType T.

Examples

```
julia> struct Foo
           x::Int64
           y::String
       end

julia> fieldtype(Foo, :x)
Int64

julia> fieldtype(Foo, 2)
String
```

source

`Base.fieldtypes` – Function.

```
fieldtypes(T::Type)
```

The declared types of all fields in a composite DataType T as a tuple.

Julia 1.1

This function requires at least Julia 1.1.

Examples

```
julia> struct Foo
           x::Int64
           y::String
       end

julia> fieldtypes(Foo)
(Int64, String)
```

source

`Base.fieldcount` – Function.

```
fieldcount(t::Type)
```

Get the number of fields that an instance of the given type would have. An error is thrown if the type is too abstract to determine this.

source

`Base.hasfield` – Function.

```
hasfield(T::Type, name::Symbol)
```

Return a boolean indicating whether T has name as one of its own fields.

See also `fieldnames`, `fieldcount`, `hasproperty`.

Julia 1.2

This function requires at least Julia 1.2.

Examples

```
julia> struct Foo
            bar::Int
       end

julia> hasfield(Foo, :bar)
true

julia> hasfield(Foo, :x)
false
```

source

Core.nfields – Function.

```
nfields(x)::Int
```

Get the number of fields in the given object.

Examples

```
julia> a = 1//2;

julia> nfields(a)
2

julia> b = 1
1

julia> nfields(b)
0

julia> ex = ErrorException("I've done a bad thing");

julia> nfields(ex)
1
```

In these examples, a is a Rational, which has two fields. b is an Int, which is a primitive bitstype with no fields at all. ex is an ErrorException, which has one field.

source

Base.isconst – Function.

```
isconst(t::DataType, s::Union{Int,Symbol})::Bool
```

Determine whether a field s is const in a given type t in the sense that a read from said field is consistent for egal objects. Note in particular that out-of-bounds fields are considered const under this definition (because they always throw).

source

```
isconst(m::Module, s::Symbol)::Bool
isconst(g::GlobalRef)::Bool
```

Determine whether a global is const in a given module m, either because it was declared constant or because it was imported from a constant binding. Note that constant-ness is specific to a particular world age, so the result of this function may not be assumed to hold after a world age update.

source

Base.isfieldatomic – Function.

```
isfieldatomic(t::DataType, s::Union{Int,Symbol})::Bool
```

Determine whether a field s is declared @atomic in a given type t.

source

Memory layout

`Base.sizeof` – Method.

```
sizeof(T::DataType)
sizeof(obj)
```

Size, in bytes, of the canonical binary representation of the given `DataType` `T`, if any. Or the size, in bytes, of object `obj` if it is not a `DataType`.

See also `Base.summarysize`.

Examples

```
julia> sizeof(Float32)
4

julia> sizeof(ComplexF64)
16

julia> sizeof(1.0)
8

julia> sizeof(collect(1.0:10.0))
80

julia> struct StructWithPadding
           x::Int64
           flag::Bool
       end

julia> sizeof(StructWithPadding) # not the sum of `sizeof` of fields due to padding
16

julia> sizeof(Int64) + sizeof(Bool) # different from above
9
```

If `DataType` `T` does not have a specific size, an error is thrown.

```
julia> sizeof(AbstractArray)
ERROR: Abstract type AbstractArray does not have a definite size.
Stacktrace:
[...]
```

source

`Base.isconcretetype` – Function.

```
isconcretetype(T)
```

Determine whether type `T` is a concrete type, meaning it could have direct instances (values `x` such that `typeof(x) === T`). Note that this is not the negation of `isabstracttype(T)`. If `T` is not a type, then return `false`.

Note

While concrete types are not abstract and vice versa, types can be neither concrete nor abstract (for example, `Vector` (a `UnionAll`)).

Note

T must be the exact type that would be returned from `typeof`. It is possible for a type U to exist such that `T == U`, `isconcretetype(T)`, but `!isconcretetype(U)`.

See also: `isbits`, `isabstracttype`, `issingletontype`.

Examples

```
julia> isconcretetype(Complex)
false

julia> isconcretetype(Complex{Float32})
true

julia> isconcretetype(Vector)
false

julia> isconcretetype(Vector{Complex})
true

julia> isconcretetype(Vector{Complex{Float32}})
true

julia> isconcretetype(Union{})
false

julia> isconcretetype(Union{Int,String})
false

julia> isconcretetype(Tuple{T} where T<:Int)
false
```

source

`Base.isbits` – Function.

```
isbits(x)
```

Return `true` if `x` is an instance of an `isbitstype` type.

source

`Base.isbitstype` – Function.

```
isbitstype(T)
```

Return `true` if type `T` is a "plain data" type, meaning it is immutable and contains no references to other values, only `primitive` types and other `isbitstype` types. Typical examples are numeric types

such as UInt8, Float64, and Complex{Float64}. This category of types is significant since they are valid as type parameters, may not track isdefined / isassigned status, and have a defined layout that is compatible with C. If T is not a type, then return false.

See also isbits, isprimitivetype, ismutable.

Examples

```
julia> isbitstype(Complex{Float64})
true

julia> isbitstype(Complex)
false
```

source

Base.fieldoffset – Function.

```
fieldoffset(type, name::Symbol | i::Integer)
```

The byte offset of a field (specified by name or index) of a type relative to its start.

Examples

```
julia> struct Foo
           x::Int64
           y::String
       end

julia> fieldoffset(Foo, 2)
0x0000000000000008

julia> fieldoffset(Foo, :x)
0x0000000000000000
```

We can use it to summarize information about a struct:

```
julia> structinfo(T) = [(fieldoffset(T,i), fieldname(T,i), fieldtype(T,i)) for i =
↪  1:fieldcount(T)];

julia> structinfo(Base.Filesystem.StatStruct)
14-element Vector{Tuple{UInt64, Symbol, Type}}:
 (0x0000000000000000, :desc, Union{RawFD, String})
 (0x0000000000000008, :device, UInt64)
 (0x0000000000000010, :inode, UInt64)
 (0x0000000000000018, :mode, UInt64)
 (0x0000000000000020, :nlink, Int64)
 (0x0000000000000028, :uid, UInt64)
 (0x0000000000000030, :gid, UInt64)
 (0x0000000000000038, :rdev, UInt64)
 (0x0000000000000040, :size, Int64)
 (0x0000000000000048, :blksize, Int64)
 (0x0000000000000050, :blocks, Int64)
 (0x0000000000000058, :mtime, Float64)
 (0x0000000000000060, :ctime, Float64)
 (0x0000000000000068, :ioerrno, Int32)
```

Julia 1.13

Specifying the field by name rather than index requires Julia 1.13 or later.

source

Base.datatype_alignment – Function.

```
Base.datatype_alignment(dt::DataType)::Int
```

Memory allocation minimum alignment for instances of this type. Can be called on any `isconcretetype`, although for Memory it will give the alignment of the elements, not the whole object.

source

Base.datatype_haspadding – Function.

```
Base.datatype_haspadding(dt::DataType)::Bool
```

Return whether the fields of instances of this type are packed in memory, with no intervening padding bits (defined as bits whose value does not impact the semantic value of the instance itself). Can be called on any `isconcretetype`.

source

Base.datatype_pointerfree – Function.

```
Base.datatype_pointerfree(dt::DataType)::Bool
```

Return whether instances of this type can contain references to gc-managed memory. Can be called on any `isconcretetype`.

source

Special values

Base.typemin – Function.

```
typemin(T)
```

The lowest value representable by the given (real) numeric DataType T.

See also: `floatmin`, `maxintfloat`, `typemax`, `eps`.

Examples

```
julia> typemin(Int8)
-128

julia> typemin(UInt32)
0x00000000

julia> typemin(Float16)
-Inf16

julia> typemin(Float32)
```

```
-Inf32

julia> floatmin(Float32)  # smallest positive finite Float32 floating point number
1.1754944f-38

julia> nextfloat(-Inf32) == -floatmax(Float32)  # equivalent ways of getting the lowest finite
↪  Float32 floating point number
true
```

source

Base.typemax – Function.

```
typemax(T)
```

The highest value representable by the given (real) numeric DataType.

See also: floatmax, maxintfloat, typemin, eps.

Examples

```
julia> typemax(Int8)
127

julia> typemax(UInt32)
0xffffffff

julia> typemax(Float64)
Inf

julia> typemax(Float32)
Inf32

julia> floatmax(Float32)  # largest positive finite Float32 floating point number
3.4028235f38
```

source

Base.floatmin – Function.

```
floatmin(T = Float64)
```

Return the smallest positive normal number representable by the floating-point type T.

See also: typemin, maxintfloat, floatmax, eps.

Examples

```
julia> floatmin(Float16)
Float16(6.104e-5)

julia> floatmin(Float32)
1.1754944f-38

julia> floatmin()
2.2250738585072014e-308
```

source

Base.floatmax – Function.

```
floatmax(T = Float64)
```

Return the largest finite number representable by the floating-point type T.

See also: typemax, maxintfloat, floatmin, eps.

Examples

```
julia> floatmax(Float16)
Float16(6.55e4)

julia> floatmax(Float32)
3.4028235f38

julia> floatmax()
1.7976931348623157e308

julia> typemax(Float64)
Inf
```

source

Base.maxintfloat – Function.

```
maxintfloat(T, S)
```

The largest consecutive integer representable in the given floating-point type T that also does not exceed the maximum integer representable by the integer type S. Equivalently, it is the minimum of maxintfloat(T) and typemax(S).

source

```
maxintfloat(T=Float64)
```

The largest consecutive integer-valued floating-point number that is exactly represented in the given floating-point type T (which defaults to Float64).

That is, maxintfloat returns the smallest positive integer-valued floating-point number n such that n+1 is *not* exactly representable in the type T.

When an Integer-type value is needed, use Integer(maxintfloat(T)).

See also: typemax, floatmax.

source

Base.eps – Method.

```
eps(::Type{T}) where T<:AbstractFloat
eps()
```

Return the *machine epsilon* of the floating point type T (`T = Float64` by default). This is defined as the gap between 1 and the next largest value representable by `typeof(one(T))`, and is equivalent to `eps(one(T))`. (Since `eps(T)` is a bound on the *relative error* of T, it is a "dimensionless" quantity like one.)

Examples

```
julia> eps()
2.220446049250313e-16

julia> eps(Float32)
1.1920929f-7

julia> 1.0 + eps()
1.0000000000000002

julia> 1.0 + eps()/2
1.0
```

More generally, for any floating-point numeric type, eps corresponds to an upper bound on the distance to the nearest floating-point complex value: if $extfl(x)$ is the closest floating-point value to a number x (e.g. an arbitrary real number), then $extfl(x)$ satisfies $|x - extfl(x)|exteps(x)/2$, not including overflow cases. This allows the definition of eps to be extended to complex numbers, for which $extfl(a + ib) = extfl(a) + iextfl(b)$.

source

Base.eps – Method.

```
eps(x::AbstractFloat)
```

Return the *unit in last place* (ulp) of x. This is the distance between consecutive representable floating point values at x. In most cases, if the distance on either side of x is different, then the larger of the two is taken, that is

```
eps(x) == max(x-prevfloat(x), nextfloat(x)-x)
```

The exceptions to this rule are the smallest and largest finite values (e.g. `nextfloat(-Inf)` and `prevfloat(Inf)` for Float64), which round to the smaller of the values.

The rationale for this behavior is that eps bounds the floating point rounding error. Under the default `RoundNearest` rounding mode, if y is a real number and x is the nearest floating point number to y, then

$$|y - x| \leq \text{eps}(x)/2.$$

See also: nextfloat, issubnormal, floatmax.

Examples

```
julia> eps(1.0)
2.220446049250313e-16
```

```
julia> eps(prevfloat(2.0))
2.220446049250313e-16

julia> eps(2.0)
4.440892098500626e-16

julia> x = prevfloat(Inf)      # largest finite Float64
1.7976931348623157e308

julia> x + eps(x)/2            # rounds up
Inf

julia> x + prevfloat(eps(x)/2) # rounds down
1.7976931348623157e308
```

source

`Base.instances` – Function.

```
instances(T::Type)
```

Return a collection of all instances of the given type, if applicable. Mostly used for enumerated types (see `@enum`).

Examples

```
julia> @enum Color red blue green

julia> instances(Color)
(red, blue, green)
```

source

44.8 Special Types

`Core.Any` – Type.

```
Any::DataType
```

Any is the union of all types. It has the defining property `isa(x, Any) == true` for any `x`. `Any` therefore describes the entire universe of possible values. For example `Integer` is a subset of `Any` that includes `Int`, `Int8`, and other integer types.

source

`Core.Union` – Type.

```
Union{Types...}
```

A `Union` type is an abstract type which includes all instances of any of its argument types. This means that `T <: Union{T,S}` and `S <: Union{T,S}`.

Like other abstract types, it cannot be instantiated, even if all of its arguments are non abstract.

Examples

```
julia> IntOrString = Union{Int,AbstractString}
Union{Int64, AbstractString}

julia> 1 isa IntOrString # instance of Int is included in the union
true

julia> "Hello!" isa IntOrString # String is also included
true

julia> 1.0 isa IntOrString # Float64 is not included because it is neither Int nor
↪  AbstractString
false
```

Extended Help

Unlike most other parametric types, unions are covariant in their parameters. For example, `Union{Real, String}` is a subtype of `Union{Number, AbstractString}`.

The empty union `Union{}` is the bottom type of Julia.

source

`Union{}` – Keyword.

```
Union{}
```

`Union{}`, the empty `Union` of types, is the *bottom* type of the type system. That is, for each `T::Type`, `Union{} <: T`. Also see the subtyping operator's documentation: `<:`.

As such, `Union{}` is also an *empty/uninhabited* type, meaning that it has no values. That is, for each `x`, `isa(x, Union{}) == false`.

`Base.Bottom` is defined as its alias and the type of `Union{}` is `Core.TypeofBottom`.

Examples

```
julia> isa(nothing, Union{})
false

julia> Union{} <: Int
true

julia> typeof(Union{}) === Core.TypeofBottom
true

julia> isa(Union{}, Union)
false
```

source

`Core.TypeofBottom` – Type.

```
Core.TypeofBottom
```

The singleton type containing only the value `Union{}` (which represents the empty type).

source

`Core.UnionAll` – Type.

```
UnionAll
```

A union of types over all values of a type parameter. `UnionAll` is used to describe parametric types where the values of some parameters are not known. See the manual section on UnionAll Types.

Examples

```
julia> typeof(Vector)
UnionAll

julia> typeof(Vector{Int})
DataType
```

source

`Core.Tuple` – Type.

```
Tuple{Types...}
```

A tuple is a fixed-length container that can hold any values of different types, but cannot be modified (it is immutable). The values can be accessed via indexing. Tuple literals are written with commas and parentheses:

```
julia> (1, 1+1)
(1, 2)

julia> (1,)
(1,)

julia> x = (0.0, "hello", 6*7)
(0.0, "hello", 42)

julia> x[2]
"hello"

julia> typeof(x)
Tuple{Float64, String, Int64}
```

A length-1 tuple must be written with a comma, `(1,)`, since `(1)` would just be a parenthesized value. `()` represents the empty (length-0) tuple.

A tuple can be constructed from an iterator by using a `Tuple` type as constructor:

```
julia> Tuple(["a", 1])
("a", 1)

julia> Tuple{String, Float64}(["a", 1])
("a", 1.0)
```

Tuple types are covariant in their parameters: `Tuple{Int}` is a subtype of `Tuple{Any}`. Therefore `Tuple{Any}` is considered an abstract type, and tuple types are only concrete if their parameters are.

Tuples do not have field names; fields are only accessed by index. Tuple types may have any number of parameters.

See the manual section on Tuple Types.

See also Vararg, NTuple, ntuple, tuple, NamedTuple.

source

Core.NTuple – Type.

```
NTuple{N, T}
```

A compact way of representing the type for a tuple of length N where all elements are of type T.

Examples

```
julia> isa((1, 2, 3, 4, 5, 6), NTuple{6, Int})
true
```

See also ntuple.

source

Core.NamedTuple – Type.

```
NamedTuple
```

NamedTuples are, as their name suggests, named Tuples. That is, they're a tuple-like collection of values, where each entry has a unique name, represented as a Symbol. Like Tuples, NamedTuples are immutable; neither the names nor the values can be modified in place after construction.

A named tuple can be created as a tuple literal with keys, e.g. `(a=1, b=2)`, or as a tuple literal with semicolon after the opening parenthesis, e.g. `(; a=1, b=2)` (this form also accepts programmatically generated names as described below), or using a NamedTuple type as constructor, e.g. `NamedTuple{(:a, :b)}((1,2))`.

Accessing the value associated with a name in a named tuple can be done using field access syntax, e.g. `x.a`, or using getindex, e.g. `x[:a]` or `x[(:a, :b)]`. A tuple of the names can be obtained using keys, and a tuple of the values can be obtained using values.

Note

Iteration over NamedTuples produces the *values* without the names. (See example below.) To iterate over the name-value pairs, use the pairs function.

The @NamedTuple macro can be used for conveniently declaring NamedTuple types.

Examples

```
julia> x = (a=1, b=2)
(a = 1, b = 2)

julia> x.a
1
```

```
julia> x[:a]
1

julia> x[(:a,)]
(a = 1,)

julia> keys(x)
(:a, :b)

julia> values(x)
(1, 2)

julia> collect(x)
2-element Vector{Int64}:
 1
 2

julia> collect(pairs(x))
2-element Vector{Pair{Symbol, Int64}}:
 :a => 1
 :b => 2
```

In a similar fashion as to how one can define keyword arguments programmatically, a named tuple can be created by giving pairs `name::Symbol => value` after a semicolon inside a tuple literal. This and the `name=value` syntax can be mixed:

```
julia> (; :a => 1, :b => 2, c=3)
(a = 1, b = 2, c = 3)
```

The name-value pairs can also be provided by splatting a named tuple or any iterator that yields two-value collections holding each a symbol as first value:

```
julia> keys = (:a, :b, :c); values = (1, 2, 3);

julia> NamedTuple{keys}(values)
(a = 1, b = 2, c = 3)

julia> (; (keys .=> values)...)
(a = 1, b = 2, c = 3)

julia> nt1 = (a=1, b=2);

julia> nt2 = (c=3, d=4);

julia> (; nt1..., nt2..., b=20) # the final b overwrites the value from nt1
(a = 1, b = 20, c = 3, d = 4)

julia> (; zip(keys, values)...) # zip yields tuples such as (:a, 1)
(a = 1, b = 2, c = 3)
```

As in keyword arguments, identifiers and dot expressions imply names:

```
julia> x = 0
```

```
0

julia> t = (; x)
(x = 0,)

julia> (; t.x)
(x = 0,)
```

Julia 1.5

Implicit names from identifiers and dot expressions are available as of Julia 1.5.

Julia 1.7

Use of getindex methods with multiple Symbols is available as of Julia 1.7.

source

Base.@NamedTuple – Macro.

```
@NamedTuple{key1::Type1, key2::Type2, ...}
@NamedTuple begin key1::Type1; key2::Type2; ...; end
```

This macro gives a more convenient syntax for declaring NamedTuple types. It returns a NamedTuple type with the given keys and types, equivalent to NamedTuple{(:key1, :key2, ...), Tuple{Type1,Type2,...}}. If the ::Type declaration is omitted, it is taken to be Any. The begin ... end form allows the declarations to be split across multiple lines (similar to a struct declaration), but is otherwise equivalent. The NamedTuple macro is used when printing NamedTuple types to e.g. the REPL.

For example, the tuple (a=3.1, b="hello") has a type NamedTuple{(:a, :b), Tuple{Float64, String}}, which can also be declared via @NamedTuple as:

```
julia> @NamedTuple{a::Float64, b::String}
@NamedTuple{a::Float64, b::String}

julia> @NamedTuple begin
           a::Float64
           b::String
       end
@NamedTuple{a::Float64, b::String}
```

Julia 1.5

This macro is available as of Julia 1.5.

source

Base.@Kwargs – Macro.

```
@Kwargs{key1::Type1, key2::Type2, ...}
```

This macro gives a convenient way to construct the type representation of keyword arguments from the same syntax as `@NamedTuple`. For example, when we have a function call like `func([positional arguments]; kw1=1.0, kw2="2")`, we can use this macro to construct the internal type representation of the keyword arguments as `@Kwargs{kw1::Float64, kw2::String}`. The macro syntax is specifically designed to simplify the signature type of a keyword method when it is printed in the stack trace view.

```
julia> @Kwargs{init::Int} # the internal representation of keyword arguments
Base.Pairs{Symbol, Int64, Nothing, @NamedTuple{init::Int64}}

julia> sum("julia"; init=1)
ERROR: MethodError: no method matching +(::Char, ::Char)
The function `+` exists, but no method is defined for this combination of argument types.

Closest candidates are:
  +(::Any, ::Any, ::Any, ::Any...)
   @ Base operators.jl:585
  +(::Integer, ::AbstractChar)
   @ Base char.jl:247
  +(::T, ::Integer) where T<:AbstractChar
   @ Base char.jl:237

Stacktrace:
  [1] add_sum(x::Char, y::Char)
    @ Base ./reduce.jl:24
  [2] BottomRF
    @ Base ./reduce.jl:86 [inlined]
  [3] _foldl_impl(op::Base.BottomRF{typeof(Base.add_sum)}, init::Int64, itr::String)
    @ Base ./reduce.jl:62
  [4] foldl_impl(op::Base.BottomRF{typeof(Base.add_sum)}, nt::Int64, itr::String)
    @ Base ./reduce.jl:48 [inlined]
  [5] mapfoldl_impl(f::typeof(identity), op::typeof(Base.add_sum), nt::Int64, itr::String)
    @ Base ./reduce.jl:44 [inlined]
  [6] mapfoldl(f::typeof(identity), op::typeof(Base.add_sum), itr::String; init::Int64)
    @ Base ./reduce.jl:175 [inlined]
  [7] mapreduce(f::typeof(identity), op::typeof(Base.add_sum), itr::String;
  ↪  kw::@Kwargs{init::Int64})
    @ Base ./reduce.jl:307 [inlined]
  [8] sum(f::typeof(identity), a::String; kw::@Kwargs{init::Int64})
    @ Base ./reduce.jl:535 [inlined]
  [9] sum(a::String; kw::@Kwargs{init::Int64})
    @ Base ./reduce.jl:564 [inlined]
 [10] top-level scope
    @ REPL[12]:1
```

Julia 1.10

This macro is available as of Julia 1.10.

source

`Base.Val` – Type.

```
Val(c)
```

Return `Val{c}()`, which contains no run-time data. Types like this can be used to pass the information between functions through the value `c`, which must be an `isbits` value or a `Symbol`. The intent of this construct is to be able to dispatch on constants directly (at compile time) without having to test the value of the constant at run time.

Examples

```
julia> f(::Val{true}) = "Good"
f (generic function with 1 method)

julia> f(::Val{false}) = "Bad"
f (generic function with 2 methods)

julia> f(Val(true))
"Good"
```

source

`Core.Vararg` – Constant.

```
Vararg{T,N}
```

The last parameter of a tuple type `Tuple` can be the special value `Vararg`, which denotes any number of trailing elements. `Vararg{T,N}` corresponds to exactly `N` elements of type `T`. Finally `Vararg{T}` corresponds to zero or more elements of type `T`. `Vararg` tuple types are used to represent the arguments accepted by varargs methods (see the section on Varargs Functions in the manual.)

See also `NTuple`.

Examples

```
julia> mytupletype = Tuple{AbstractString, Vararg{Int}}
Tuple{AbstractString, Vararg{Int64}}

julia> isa(("1",), mytupletype)
true

julia> isa(("1",1), mytupletype)
true

julia> isa(("1",1,2), mytupletype)
true

julia> isa(("1",1,2,3.0), mytupletype)
false
```

source

`Core.Nothing` – Type.

```
Nothing
```

A type with no fields that is the type of `nothing`.

See also: `isnothing`, `Some`, `Missing`.

source

`Base.isnothing` – Function.

```
isnothing(x)
```

Return `true` if `x === nothing`, and return `false` if not.

Julia 1.1

This function requires at least Julia 1.1.

See also `something`, `Base.notnothing`, `ismissing`.

source

`Base.notnothing` – Function.

```
notnothing(x)
```

Throw an error if `x === nothing`, and return `x` if not.

source

`Base.Some` – Type.

```
Some{T}
```

A wrapper type used in `Union{Some{T}, Nothing}` to distinguish between the absence of a value (`nothing`) and the presence of a `nothing` value (i.e. `Some(nothing)`).

Use `something` to access the value wrapped by a `Some` object.

source

`Base.something` – Function.

```
something(x...)
```

Return the first value in the arguments which is not equal to `nothing`, if any. Otherwise throw an error. Arguments of type `Some` are unwrapped.

See also `coalesce`, `skipmissing`, `@something`.

Examples

```
julia> something(nothing, 1)
1

julia> something(Some(1), nothing)
1

julia> something(Some(nothing), 2) === nothing
```

```
true

julia> something(missing, nothing)
missing

julia> something(nothing, nothing)
ERROR: ArgumentError: No value arguments present
```

source

Base.@something – Macro.

```
@something(x...)
```

Short-circuiting version of something.

Examples

```
julia> f(x) = (println("f($x)"); nothing);

julia> a = 1;

julia> a = @something a f(2) f(3) error("Unable to find default for `a`")
1

julia> b = nothing;

julia> b = @something b f(2) f(3) error("Unable to find default for `b`")
f(2)
f(3)
ERROR: Unable to find default for `b`
[...]

julia> b = @something b f(2) f(3) Some(nothing)
f(2)
f(3)

julia> b === nothing
true
```

Julia 1.7

This macro is available as of Julia 1.7.

source

Base.Enums.Enum – Type.

```
Enum{T<:Integer}
```

The abstract supertype of all enumerated types defined with @enum.

source

Base.Enums.@enum – Macro.

```
@enum EnumName[::BaseType] value1[=x] value2[=y]
```

Create an `Enum{BaseType}` subtype with name `EnumName` and enum member values of `value1` and `value2` with optional assigned values of `x` and `y`, respectively. `EnumName` can be used just like other types and enum member values as regular values, such as

Examples

```
julia> @enum Fruit apple=1 orange=2 kiwi=3

julia> f(x::Fruit) = "I'm a Fruit with value: $(Int(x))"
f (generic function with 1 method)

julia> f(apple)
"I'm a Fruit with value: 1"

julia> Fruit(1)
apple::Fruit = 1
```

Values can also be specified inside a `begin` block, e.g.

```
@enum EnumName begin
    value1
    value2
end
```

`BaseType`, which defaults to `Int32`, must be a primitive subtype of `Integer`. Member values can be converted between the enum type and `BaseType`. `read` and `write` perform these conversions automatically. In case the enum is created with a non-default `BaseType`, `Integer(value1)` will return the integer `value1` with the type `BaseType`.

To list all the instances of an enum use `instances`, e.g.

```
julia> instances(Fruit)
(apple, orange, kiwi)
```

It is possible to construct a symbol from an enum instance:

```
julia> Symbol(apple)
:apple
```

source

`Core.Expr` – Type.

```
Expr(head::Symbol, args...)
```

A type representing compound expressions in parsed julia code (ASTs). Each expression consists of a `head` `Symbol` identifying which kind of expression it is (e.g. a call, for loop, conditional statement, etc.), and subexpressions (e.g. the arguments of a call). The subexpressions are stored in a `Vector{Any}` field called `args`.

See the manual chapter on Metaprogramming and the developer documentation Julia ASTs.

Examples

```
julia> Expr(:call, :+, 1, 2)
:(1 + 2)

julia> dump(:(a ? b : c))
Expr
  head: Symbol if
  args: Array{Any}((3,))
    1: Symbol a
    2: Symbol b
    3: Symbol c
```

source

`Core.Symbol` – Type.

```
Symbol
```

The type of object used to represent identifiers in parsed julia code (ASTs). Also often used as a name or label to identify an entity (e.g. as a dictionary key). `Symbol`s can be entered using the `:` quote operator:

```
julia> :name
:name

julia> typeof(:name)
Symbol

julia> x = 42
42

julia> eval(:x)
42
```

`Symbol`s can also be constructed from strings or other values by calling the constructor `Symbol(x...)`.

`Symbol`s are immutable and their implementation re-uses the same object for all `Symbol`s with the same name.

Unlike strings, `Symbol`s are "atomic" or "scalar" entities that do not support iteration over characters.

source

`Core.Symbol` – Method.

```
Symbol(x...)::Symbol
```

Create a `Symbol` by concatenating the string representations of the arguments together.

Examples

```
julia> Symbol("my", "name")
:myname

julia> Symbol("day", 4)
:day4
```

source

`Core.Module` – Type.

```
Module
```

A `Module` is a separate global variable workspace. See `module` and the manual section about modules for details.

```
Module(name::Symbol=:anonymous, std_imports=true, default_names=true)
```

Return a module with the specified name. A `baremodule` corresponds to `Module(:ModuleName, false)`

An empty module containing no names at all can be created with `Module(:ModuleName, false, false)`. This module will not import `Base` or `Core` and does not contain a reference to itself.

source

44.9 Generic Functions

`Core.Function` – Type.

```
Function
```

Abstract type of all functions.

Examples

```
julia> isa(+, Function)
true

julia> typeof(sin)
typeof(sin) (singleton type of function sin, subtype of Function)

julia> ans <: Function
true
```

source

`Base.hasmethod` – Function.

```
hasmethod(f, t::Type{<:Tuple}[, kwnames]; world=get_world_counter())::Bool
```

Determine whether the given generic function has a method matching the given `Tuple` of argument types with the upper bound of world age given by `world`.

If a tuple of keyword argument names `kwnames` is provided, this also checks whether the method of `f` matching `t` has the given keyword argument names. If the matching method accepts a variable number of keyword arguments, e.g. with `kwargs...`, any names given in `kwnames` are considered valid. Otherwise the provided names must be a subset of the method's keyword arguments.

See also `applicable`.

Julia 1.2

Providing keyword argument names requires Julia 1.2 or later.

Examples

```
julia> hasmethod(length, Tuple{Array})
true

julia> f(; oranges=0) = oranges;

julia> hasmethod(f, Tuple{}, (:oranges,))
true

julia> hasmethod(f, Tuple{}, (:apples, :bananas))
false

julia> g(; xs...) = 4;

julia> hasmethod(g, Tuple{}, (:a, :b, :c, :d))  # g accepts arbitrary kwargs
true
```

source

`Core.applicable` – Function.

```
applicable(f, args...)::Bool
```

Determine whether the given generic function has a method applicable to the given arguments.

See also `hasmethod`.

Examples

```
julia> function f(x, y)
           x + y
       end;

julia> applicable(f, 1)
false

julia> applicable(f, 1, 2)
true
```

source

`Base.isambiguous` – Function.

```
Base.isambiguous(m1, m2; ambiguous_bottom=false)::Bool
```

Determine whether two methods `m1` and `m2` may be ambiguous for some call signature. This test is performed in the context of other methods of the same function; in isolation, `m1` and `m2` might be ambiguous, but if a third method resolving the ambiguity has been defined, this returns `false`. Alternatively, in isolation `m1` and `m2` might be ordered, but if a third method cannot be sorted with them, they may cause an ambiguity together.

For parametric types, the `ambiguous_bottom` keyword argument controls whether `Union{}` counts as an ambiguous intersection of type parameters – when `true`, it is considered ambiguous, when `false` it is not.

Examples

```
julia> foo(x::Complex{<:Integer}) = 1
foo (generic function with 1 method)

julia> foo(x::Complex{<:Rational}) = 2
foo (generic function with 2 methods)

julia> m1, m2 = collect(methods(foo));

julia> typeintersect(m1.sig, m2.sig)
Tuple{typeof(foo), Complex{Union{}}}

julia> Base.isambiguous(m1, m2, ambiguous_bottom=true)
true

julia> Base.isambiguous(m1, m2, ambiguous_bottom=false)
false
```

source

`Core.invoke` – Function.

```
invoke(f, argtypes::Type, args...; kwargs...)
invoke(f, argtypes::Method, args...; kwargs...)
invoke(f, argtypes::CodeInstance, args...; kwargs...)
```

Invoke a method for the given generic function `f` matching the specified types `argtypes` on the specified arguments `args` and passing the keyword arguments `kwargs`. The arguments `args` must conform with the specified types in `argtypes`, i.e. conversion is not automatically performed. This method allows invoking a method other than the most specific matching method, which is useful when the behavior of a more general definition is explicitly needed (often as part of the implementation of a more specific method of the same function). However, because this means the runtime must do more work, `invoke` is generally also slower–sometimes significantly so–than doing normal dispatch with a regular call.

Be careful when using `invoke` for functions that you don't write. What definition is used for given `argtypes` is an implementation detail unless the function is explicitly states that calling with certain `argtypes` is a part of public API. For example, the change between `f1` and `f2` in the example below is usually considered compatible because the change is invisible by the caller with a normal (non-`invoke`) call. However, the change is visible if you use `invoke`.

Passing a `Method` instead of a signature

The `argtypes` argument may be a `Method`, in which case the ordinary method table lookup is bypassed entirely and the given method is invoked directly. Needing this feature is uncommon. Note in particular that the specified `Method` may be entirely unreachable from ordinary dispatch (or ordinary invoke), e.g. because it was replaced or fully covered by more specific methods. If the method is part of the ordinary method table, this call behaves similar to `invoke(f, method.sig, args...)`.

Julia 1.12

Passing a `Method` requires Julia 1.12.

Passing a `CodeInstance` instead of a signature

The `argtypes` argument may be a `CodeInstance`, bypassing both method lookup and specialization. The semantics of this invocation are similar to a function pointer call of the `CodeInstance`'s `invoke` pointer. It is an error to invoke a `CodeInstance` with arguments that do not match its parent `MethodInstance` or from a world age not included in the `min_world`/`max_world` range. It is undefined behavior to invoke a `CodeInstance` whose behavior does not match the constraints specified in its fields. For some code instances with `owner !== nothing` (i.e. those generated by external compilers), it may be an error to invoke them after passing through precompilation. This is an advanced interface intended for use with external compiler plugins.

Julia 1.12

Passing a `CodeInstance` requires Julia 1.12.

Examples

```
julia> f(x::Real) = x^2;

julia> f(x::Integer) = 1 + invoke(f, Tuple{Real}, x);

julia> f(2)
5

julia> f1(::Integer) = Integer
       f1(::Real) = Real;

julia> f2(x::Real) = _f2(x)
       _f2(::Integer) = Integer
       _f2(_) = Real;

julia> f1(1)
Integer

julia> f2(1)
Integer

julia> invoke(f1, Tuple{Real}, 1)
Real

julia> invoke(f2, Tuple{Real}, 1)
Integer
```

source

Base.@invoke – Macro.

```
@invoke f(arg::T, ...; kwargs...)
```

Provides a convenient way to call `invoke` by expanding `@invoke f(arg1::T1, arg2::T2; kwargs...)` to `invoke(f, Tuple{T1,T2}, arg1, arg2; kwargs...)`. When an argument's type annotation is omitted, it's replaced with `Core.Typeof` that argument. To invoke a method where an argument is untyped or explicitly typed as `Any`, annotate the argument with `::Any`.

It also supports the following syntax:

- `@invoke (x::X).f` expands to `invoke(getproperty, Tuple{X,Symbol}, x, :f)`
- `@invoke (x::X).f = v::V` expands to `invoke(setproperty!, Tuple{X,Symbol,V}, x, :f, v)`
- `@invoke (xs::Xs)[i::I]` expands to `invoke(getindex, Tuple{Xs,I}, xs, i)`
- `@invoke (xs::Xs)[i::I] = v::V` expands to `invoke(setindex!, Tuple{Xs,V,I}, xs, v, i)`

Examples

```
julia> @macroexpand @invoke f(x::T, y)
:(Core.invoke(f, Base.Tuple{T, Core.Typeof(y)}, x, y))

julia> @invoke 420::Integer % Unsigned
0x00000000000001a4

julia> @macroexpand @invoke (x::X).f
:(Core.invoke(Base.getproperty, Base.Tuple{X, Core.Typeof(:f)}, x, :f))

julia> @macroexpand @invoke (x::X).f = v::V
:(Core.invoke(Base.setproperty!, Base.Tuple{X, Core.Typeof(:f), V}, x, :f, v))

julia> @macroexpand @invoke (xs::Xs)[i::I]
:(Core.invoke(Base.getindex, Base.Tuple{Xs, I}, xs, i))

julia> @macroexpand @invoke (xs::Xs)[i::I] = v::V
:(Core.invoke(Base.setindex!, Base.Tuple{Xs, V, I}, xs, v, i))
```

Julia 1.7

This macro requires Julia 1.7 or later.

Julia 1.9

This macro is exported as of Julia 1.9.

Julia 1.10

The additional syntax is supported as of Julia 1.10.

source

`Base.invokelatest` – Function.

```
invokelatest(f, args...; kwargs...)
```

Calls `f(args...; kwargs...)`, but guarantees that the most recent method of `f` will be executed. This is useful in specialized circumstances, e.g. long-running event loops or callback functions that may call obsolete versions of a function `f`. (The drawback is that `invokelatest` is somewhat slower than calling `f` directly, and the type of the result cannot be inferred by the compiler.)

Julia 1.9

Prior to Julia 1.9, this function was not exported, and was called as `Base.invokelatest`.

source

`Base.@invokelatest` – Macro.

```
@invokelatest f(args...; kwargs...)
```

Provides a convenient way to call `invokelatest`. `@invokelatest f(args...; kwargs...)` will simply be expanded into `Base.invokelatest(f, args...; kwargs...)`.

It also supports the following syntax:

- `@invokelatest x.f` expands to `Base.invokelatest(getproperty, x, :f)`
- `@invokelatest x.f = v` expands to `Base.invokelatest(setproperty!, x, :f, v)`
- `@invokelatest xs[i]` expands to `Base.invokelatest(getindex, xs, i)`
- `@invokelatest xs[i] = v` expands to `Base.invokelatest(setindex!, xs, v, i)`

Note

If `f` is a global, it will be resolved consistently in the (latest) world as the call target. However, all other arguments (as well as `f` itself if it is not a literal global) will be evaluated in the current world age.

Julia 1.7

This macro requires Julia 1.7 or later.

Julia 1.9

Prior to Julia 1.9, this macro was not exported, and was called as `Base.@invokelatest`.

Julia 1.10

The additional `x.f` and `xs[i]` syntax requires Julia 1.10.

source

`new` – Keyword.

```
new, or new{A,B,...}
```

Special function available to inner constructors which creates a new object of the type. The form new{A,B,...} explicitly specifies values of parameters for parametric types. See the manual section on Inner Constructor Methods for more information.

source

Base.:|> – Function.

```
|>(x, f)
```

Infix operator which applies function f to the argument x. This allows f(g(x)) to be written x |> g |> f. When used with anonymous functions, parentheses are typically required around the definition to get the intended chain.

Examples

```
julia> 4 |> inv
0.25

julia> [2, 3, 5] |> sum |> inv
0.1

julia> [0 1; 2 3] .|> (x -> x^2) |> sum
14
```

source

Base.:∘ – Function.

```
f ∘ g
```

Compose functions: i.e. (f ∘ g)(args...; kwargs...) means f(g(args...; kwargs...)). The ∘ symbol can be entered in the Julia REPL (and most editors, appropriately configured) by typing \circ<tab>.

Function composition also works in prefix form: ∘(f, g) is the same as f ∘ g. The prefix form supports composition of multiple functions: ∘(f, g, h) = f ∘ g ∘ h and splatting ∘(fs...) for composing an iterable collection of functions. The last argument to ∘ execute first.

Julia 1.4

Multiple function composition requires at least Julia 1.4.

Julia 1.5

Composition of one function ∘(f) requires at least Julia 1.5.

Julia 1.7

Using keyword arguments requires at least Julia 1.7.

Examples

```
julia> map(uppercase∘first, ["apple", "banana", "carrot"])
3-element Vector{Char}:
 'A': ASCII/Unicode U+0041 (category Lu: Letter, uppercase)
 'B': ASCII/Unicode U+0042 (category Lu: Letter, uppercase)
 'C': ASCII/Unicode U+0043 (category Lu: Letter, uppercase)

julia> (==(6)∘length).(["apple", "banana", "carrot"])
3-element BitVector:
 0
 1
 1

julia> fs = [
           x -> 2x
           x -> x-1
           x -> x/2
           x -> x+1
       ];

julia> ∘(fs...)(3)
2.0
```

See also `ComposedFunction`, `!f::Function`.

source

`Base.ComposedFunction` – Type.

```
ComposedFunction{Outer,Inner} <: Function
```

Represents the composition of two callable objects `outer::Outer` and `inner::Inner`. That is

```
ComposedFunction(outer, inner)(args...; kw...) === outer(inner(args...; kw...))
```

The preferred way to construct an instance of `ComposedFunction` is to use the composition operator `∘`:

```
julia> sin ∘ cos === ComposedFunction(sin, cos)
true

julia> typeof(sin∘cos)
ComposedFunction{typeof(sin), typeof(cos)}
```

The composed pieces are stored in the fields of `ComposedFunction` and can be retrieved as follows:

```
julia> composition = sin ∘ cos
sin ∘ cos

julia> composition.outer === sin
true

julia> composition.inner === cos
true
```

Julia 1.6

ComposedFunction requires at least Julia 1.6. In earlier versions ∘ returns an anonymous function instead.

See also ∘.

source

Base.splat – Function.

```
splat(f)
```

Equivalent to

```
    my_splat(f) = args->f(args...)
```

i.e. given a function returns a new function that takes one argument and splats it into the original function. This is useful as an adaptor to pass a multi-argument function in a context that expects a single argument, but passes a tuple as that single argument.

Examples

```
julia> map(splat(+), zip(1:3,4:6))
3-element Vector{Int64}:
 5
 7
 9

julia> my_add = splat(+)
splat(+)

julia> my_add((1,2,3))
6
```

source

Base.Fix – Type.

```
Fix{N}(f, x)
```

A type representing a partially-applied version of a function `f`, with the argument `x` fixed at position `N::Int`. In other words, `Fix{3}(f, x)` behaves similarly to `(y1, y2, y3...; kws...) -> f(y1, y2, x, y3...; kws...)`.

Julia 1.12

This general functionality requires at least Julia 1.12, while `Fix1` and `Fix2` are available earlier.

Note

When nesting multiple `Fix`, note that the `N` in `Fix{N}` is *relative* to the current available arguments, rather than an absolute ordering on the target function. For example, `Fix{1}(Fix{2}(f, 4), 4)` fixes the first and second arg, while `Fix{2}(Fix{1}(f, 4), 4)` fixes the first and third arg.

source

Base.Fix1 – Type.

Alias for Fix{1}. See Fix.

source

Base.Fix2 – Type.

Alias for Fix{2}. See Fix.

source

Base.Returns – Type.

```
f = Returns(value)
```

Create a callable f such that f(args...; kw...) === value holds.

Examples

```
julia> f = Returns(42);

julia> f(1)
42

julia> f("hello", x=32)
42

julia> f.value
42
```

Julia 1.7

Returns requires at least Julia 1.7.

source

44.10 Syntax

Core.eval – Function.

```
Core.eval(m::Module, expr)
```

Evaluate an expression in the given module and return the result.

source

eval – Function.

```
eval(expr)
```

Evaluate an expression in the global scope of the containing module. Every `Module` (except those defined with `baremodule`) has a private 1-argument definition of `eval`, which evaluates expressions in that module, for use inside that module.

source

`Base.@eval` – Macro.

```
@eval [mod,] ex
```

Evaluate an expression with values interpolated into it using `eval`. If two arguments are provided, the first is the module to evaluate in.

source

`Base.evalfile` – Function.

```
evalfile(path::AbstractString, args::Vector{String}=String[])
```

Load the file into an anonymous module using `include`, evaluate all expressions, and return the value of the last expression. The optional `args` argument can be used to set the input arguments of the script (i.e. the global ARGS variable). Note that definitions (e.g. methods, globals) are evaluated in the anonymous module and do not affect the current module.

Examples

```
julia> write("testfile.jl", """
           @show ARGS
           1 + 1
       """);

julia> x = evalfile("testfile.jl", ["ARG1", "ARG2"]);
ARGS = ["ARG1", "ARG2"]

julia> x
2

julia> rm("testfile.jl")
```

source

`Base.esc` – Function.

```
esc(e)
```

Only valid in the context of an `Expr` returned from a macro. Prevents the macro hygiene pass from turning embedded variables into gensym variables. See the Macros section of the Metaprogramming chapter of the manual for more details and examples.

source

`Base.@inbounds` – Macro.

```
@inbounds(blk)
```

Eliminates array bounds checking within expressions.

In the example below the in-range check for referencing element `i` of array `A` is skipped to improve performance.

```
function sum(A::AbstractArray)
    r = zero(eltype(A))
    for i in eachindex(A)
        @inbounds r += A[i]
    end
    return r
end
```

Warning

Using `@inbounds` may return incorrect results/crashes/corruption for out-of-bounds indices. The user is responsible for checking it manually. Only use `@inbounds` when you are certain that all accesses are in bounds (as undefined behavior, e.g. crashes, might occur if this assertion is violated). For example, using `1:length(A)` instead of `eachindex(A)` in a function like the one above is *not* safely inbounds because the first index of A may not be 1 for all user defined types that subtype `AbstractArray`.

source

`Base.@boundscheck` – Macro.

```
@boundscheck(blk)
```

Annotates the expression `blk` as a bounds checking block, allowing it to be elided by `@inbounds`.

Note

The function in which `@boundscheck` is written must be inlined into its caller in order for `@inbounds` to have effect.

Examples

```
julia> @inline function g(A, i)
           @boundscheck checkbounds(A, i)
           return "accessing ($A)[$i]"
       end;

julia> f1() = return g(1:2, -1);

julia> f2() = @inbounds return g(1:2, -1);

julia> f1()
ERROR: BoundsError: attempt to access 2-element UnitRange{Int64} at index [-1]
Stacktrace:
 [1] throw_boundserror(::UnitRange{Int64}, ::Tuple{Int64}) at ./abstractarray.jl:455
 [2] checkbounds at ./abstractarray.jl:420 [inlined]
 [3] g at ./none:2 [inlined]
 [4] f1() at ./none:1
```

```
 [5] top-level scope

julia> f2()
"accessing (1:2)[-1]"
```

Warning

The `@boundscheck` annotation allows you, as a library writer, to opt-in to allowing *other code* to remove your bounds checks with `@inbounds`. As noted there, the caller must verify—using information they can access—that their accesses are valid before using `@inbounds`. For indexing into your `AbstractArray` subclasses, for example, this involves checking the indices against its `axes`. Therefore, `@boundscheck` annotations should only be added to a `getindex` or `setindex!` implementation after you are certain its behavior is correct.

source

`Base.@propagate_inbounds` – Macro.

```
@propagate_inbounds
```

Tells the compiler to inline a function while retaining the caller's inbounds context.

source

`Base.@inline` – Macro.

```
@inline
```

Give a hint to the compiler that this function is worth inlining.

Small functions typically do not need the `@inline` annotation, as the compiler does it automatically. By using `@inline` on bigger functions, an extra nudge can be given to the compiler to inline it.

`@inline` can be applied immediately before a function definition or within a function body.

```
# annotate long-form definition
@inline function longdef(x)
    ...
end

# annotate short-form definition
@inline shortdef(x) = ...

# annotate anonymous function that a `do` block creates
f() do
    @inline
    ...
end
```

Julia 1.8

The usage within a function body requires at least Julia 1.8.

```
@inline block
```

Give a hint to the compiler that calls within `block` are worth inlining.

```
# The compiler will try to inline `f`
@inline f(...)

# The compiler will try to inline `f`, `g` and `+`
@inline f(...) + g(...)
```

Note

A callsite annotation always has the precedence over the annotation applied to the definition of the called function:

```
@noinline function explicit_noinline(args...)
    # body
end

let
    @inline explicit_noinline(args...) # will be inlined
end
```

Note

The callsite annotation applies to all calls in the block, including function arguments that are themselves calls:

```
# The compiler will not inline `getproperty`, `g` or `f`
@noinline f(x.inner, g(y))
```

Note

When there are nested callsite annotations, the innermost annotation has the precedence:

```
@noinline let a0, b0 = ...
    a = @inline f(a0)  # the compiler will try to inline this call
    b = f(b0)          # the compiler will NOT try to inline this call
    return a, b
end
```

Warning

Although a callsite annotation will try to force inlining in regardless of the cost model, there are still chances it can't succeed in it. Especially, recursive calls can not be inlined even if they are annotated as `@inlined`.

Julia 1.8

The callsite annotation requires at least Julia 1.8.

source

`Base.@noinline` – Macro.

```
@noinline
```

Give a hint to the compiler that it should not inline a function.

Small functions are typically inlined automatically. By using `@noinline` on small functions, auto-inlining can be prevented.

`@noinline` can be applied immediately before a function definition or within a function body.

```
# annotate long-form definition
@noinline function longdef(x)
    ...
end

# annotate short-form definition
@noinline shortdef(x) = ...

# annotate anonymous function that a `do` block creates
f() do
    @noinline
    ...
end
```

Julia 1.8

The usage within a function body requires at least Julia 1.8.

```
@noinline block
```

Give a hint to the compiler that it should not inline the calls within `block`.

```
# The compiler will try to not inline `f`
@noinline f(...)

# The compiler will try to not inline `f`, `g` and `+`
@noinline f(...) + g(...)
```

Note

A callsite annotation always has the precedence over the annotation applied to the definition of the called function:

```
@inline function explicit_inline(args...)
    # body
end

let
    @noinline explicit_inline(args...) # will not be inlined
end
```

Note

When there are nested callsite annotations, the innermost annotation has the precedence:

```
@inline let a0, b0 = ...
    a = @noinline f(a0)  # the compiler will NOT try to inline this call
    b = f(b0)            # the compiler will try to inline this call
    return a, b
end
```

Julia 1.8

The callsite annotation requires at least Julia 1.8.

Note

If the function is trivial (for example returning a constant) it might get inlined anyway.

source

`Base.@nospecialize` – Macro.

```
@nospecialize
```

Applied to a function argument name, hints to the compiler that the method implementation should not be specialized for different types of that argument, but instead use the declared type for that argument. It can be applied to an argument within a formal argument list, or in the function body. When applied to an argument, the macro must wrap the entire argument expression, e.g., `@nospecialize(x::Real)` or `@nospecialize(i::Integer...)` rather than wrapping just the argument name. When used in a function body, the macro must occur in statement position and before any code.

When used without arguments, it applies to all arguments of the parent scope. In local scope, this means all arguments of the containing function. In global (top-level) scope, this means all methods subsequently defined in the current module.

Specialization can reset back to the default by using `@specialize`.

```
function example_function(@nospecialize x)
    ...
end

function example_function(x, @nospecialize(y = 1))
    ...
end

function example_function(x, y, z)
    @nospecialize x y
    ...
end

@nospecialize
f(y) = [x for x in y]
@specialize
```

Note

@nospecialize affects code generation but not inference: it limits the diversity of the resulting native code, but it does not impose any limitations (beyond the standard ones) on type-inference. Use Base.@nospecializeinfer together with @nospecialize to additionally suppress inference.

Examples

```
julia> f(A::AbstractArray) = g(A)
f (generic function with 1 method)

julia> @noinline g(@nospecialize(A::AbstractArray)) = A[1]
g (generic function with 1 method)

julia> @code_typed f([1.0])
CodeInfo(
1 — %1 =    invoke g(A::AbstractArray)::Float64
└──      return %1
) => Float64
```

Here, the @nospecialize annotation results in the equivalent of

```
f(A::AbstractArray) = invoke(g, Tuple{AbstractArray}, A)
```

ensuring that only one version of native code will be generated for g, one that is generic for any AbstractArray. However, the specific return type is still inferred for both g and f, and this is still used in optimizing the callers of f and g.

source

Base.@specialize – Macro.

```
@specialize
```

Reset the specialization hint for an argument back to the default. For details, see `@nospecialize`.

source

`Base.@nospecializeinfer` – Macro.

```
Base.@nospecializeinfer function f(args...)
    @nospecialize ...
    ...
end
Base.@nospecializeinfer f(@nospecialize args...) = ...
```

Tells the compiler to infer `f` using the declared types of `@nospecialized` arguments. This can be used to limit the number of compiler-generated specializations during inference.

Examples

```
julia> f(A::AbstractArray) = g(A)
f (generic function with 1 method)

julia> @noinline Base.@nospecializeinfer g(@nospecialize(A::AbstractArray)) = A[1]
g (generic function with 1 method)

julia> @code_typed f([1.0])
CodeInfo(
1 — %1 =    invoke g(A::AbstractArray)::Any
└──      return %1
) => Any
```

In this example, `f` will be inferred for each specific type of `A`, but `g` will only be inferred once with the declared argument type `A::AbstractArray`, meaning that the compiler will not likely see the excessive inference time on it while it can not infer the concrete return type of it. Without the `@nospecializeinfer`, `f([1.0])` would infer the return type of `g` as `Float64`, indicating that inference ran for `g(::Vector{Float64})` despite the prohibition on specialized code generation.

Julia 1.10

Using `Base.@nospecializeinfer` requires Julia version 1.10.

source

`Base.@constprop` – Macro.

```
Base.@constprop setting [ex]
```

Control the mode of interprocedural constant propagation for the annotated function.

Two `setting`s are supported:

- `Base.@constprop :aggressive [ex]`: apply constant propagation aggressively. For a method where the return type depends on the value of the arguments, this can yield improved inference results at the cost of additional compile time.
- `Base.@constprop :none [ex]`: disable constant propagation. This can reduce compile times for functions that Julia might otherwise deem worthy of constant-propagation. Common cases are for functions with `Bool`- or `Symbol`-valued arguments or keyword arguments.

Base.@constprop can be applied immediately before a function definition or within a function body.

```
# annotate long-form definition
Base.@constprop :aggressive function longdef(x)
    ...
end

# annotate short-form definition
Base.@constprop :aggressive shortdef(x) = ...

# annotate anonymous function that a `do` block creates
f() do
    Base.@constprop :aggressive
    ...
end
```

Julia 1.10

The usage within a function body requires at least Julia 1.10.

source

Base.gensym – Function.

```
gensym([tag])
```

Generates a symbol which will not conflict with other variable names (in the same module).

source

Base.@gensym – Macro.

```
@gensym
```

Generates a gensym symbol for a variable. For example, `@gensym x y` is transformed into `x = gensym("x"); y = gensym("y")`.

source

var"name" – Keyword.

```
var
```

The syntax `var"#example#"` refers to a variable named `Symbol("#example#")`, even though `#example#` is not a valid Julia identifier name.

This can be useful for interoperability with programming languages which have different rules for the construction of valid identifiers. For example, to refer to the R variable `draw.segments`, you can use `var"draw.segments"` in your Julia code.

It is also used to show julia source code which has gone through macro hygiene or otherwise contains variable names which can't be parsed normally.

Note that this syntax requires parser support so it is expanded directly by the parser rather than being implemented as a normal string macro `@var_str`.

Julia 1.3

This syntax requires at least Julia 1.3.

source

Base.@goto – Macro.

```
@goto name
```

`@goto name` unconditionally jumps to the statement at the location `@label name`.

`@label` and `@goto` cannot create jumps to different top-level statements. Attempts cause an error. To still use `@goto`, enclose the `@label` and `@goto` in a block.

source

Base.@label – Macro.

```
@label name
```

Labels a statement with the symbolic label `name`. The label marks the end-point of an unconditional jump with `@goto name`.

source

Base.SimdLoop.@simd – Macro.

```
@simd
```

Annotate a `for` loop to allow the compiler to take extra liberties to allow loop re-ordering

Warning

This feature is experimental and could change or disappear in future versions of Julia. Incorrect use of the `@simd` macro may cause unexpected results.

The object iterated over in a `@simd for` loop should be a one-dimensional range. By using `@simd`, you are asserting several properties of the loop:

- It is safe to execute iterations in arbitrary or overlapping order, with special consideration for reduction variables.
- Floating-point operations on reduction variables can be reordered or contracted, possibly causing different results than without `@simd`.

In many cases, Julia is able to automatically vectorize inner for loops without the use of `@simd`. Using `@simd` gives the compiler a little extra leeway to make it possible in more situations. In either case, your inner loop should have the following properties to allow vectorization:

- The loop must be an innermost loop
- The loop body must be straight-line code. Therefore, `@inbounds` is currently needed for all array accesses. The compiler can sometimes turn short `&&`, `||`, and `?:` expressions into straight-line code if it is safe to evaluate all operands unconditionally. Consider using the `ifelse` function instead of `?:` in the loop if it is safe to do so.

- Accesses must have a stride pattern and cannot be "gathers" (random-index reads) or "scatters" (random-index writes).
- The stride should be unit stride.

Note

The `@simd` does not assert by default that the loop is completely free of loop-carried memory dependencies, which is an assumption that can easily be violated in generic code. If you are writing non-generic code, you can use `@simd ivdep for ... end` to also assert that:

- There exists no loop-carried memory dependencies
- No iteration ever waits on a previous iteration to make forward progress.

source

`Base.@polly` – Macro.

```
@polly
```

Tells the compiler to apply the polyhedral optimizer Polly to a function.

source

`Base.@generated` – Macro.

```
@generated f
```

`@generated` is used to annotate a function which will be generated. In the body of the generated function, only types of arguments can be read (not the values). The function returns a quoted expression evaluated when the function is called. The `@generated` macro should not be used on functions mutating the global scope or depending on mutable elements.

See Metaprogramming for further details.

Examples

```
julia> @generated function bar(x)
           if x <: Integer
               return :(x ^ 2)
           else
               return :(x)
           end
       end
bar (generic function with 1 method)

julia> bar(4)
16

julia> bar("baz")
"baz"
```

source

`Base.@assume_effects` – Macro.

```
Base.@assume_effects setting... [ex]
```

Override the compiler's effect modeling. This macro can be used in several contexts:

1. Immediately before a method definition, to override the entire effect modeling of the applied method.
2. Within a function body without any arguments, to override the entire effect modeling of the enclosing method.
3. Applied to a code block, to override the local effect modeling of the applied code block.

Examples

```
julia> Base.@assume_effects :terminates_locally function fact(x)
           # usage 1:
           # this :terminates_locally allows `fact` to be constant-folded
           res = 1
           0 ≤ x < 20 || error("bad fact")
           while x > 1
               res *= x
               x -= 1
           end
           return res
       end
fact (generic function with 1 method)

julia> code_typed() do
           fact(12)
       end |> only
CodeInfo(
1 —     return 479001600
) => Int64

julia> code_typed() do
           map((2,3,4)) do x
               # usage 2:
               # this :terminates_locally allows this anonymous function to be constant-folded
               Base.@assume_effects :terminates_locally
               res = 1
               0 ≤ x < 20 || error("bad fact")
               while x > 1
                   res *= x
                   x -= 1
               end
               return res
           end
       end |> only
CodeInfo(
1 —     return (2, 6, 24)
) => Tuple{Int64, Int64, Int64}
```

```
julia> code_typed() do
           map((2,3,4)) do x
               res = 1
               0 ≤ x < 20 || error("bad fact")
               # usage 3:
               # with this :terminates_locally annotation the compiler skips tainting
               # `:terminates` effect within this `while` block, allowing the parent
               # anonymous function to be constant-folded
               Base.@assume_effects :terminates_locally while x > 1
                   res *= x
                   x -= 1
               end
               return res
           end
       end |> only
CodeInfo(
1 —     return (2, 6, 24)
) => Tuple{Int64, Int64, Int64}
```

Julia 1.8

Using `Base.@assume_effects` requires Julia version 1.8.

Julia 1.10

The usage within a function body requires at least Julia 1.10.

Julia 1.11

The code block annotation requires at least Julia 1.11.

Warning

Improper use of this macro causes undefined behavior (including crashes, incorrect answers, or other hard to track bugs). Use with care and only as a last resort if absolutely required. Even in such a case, you SHOULD take all possible steps to minimize the strength of the effect assertion (e.g., do not use `:total` if `:nothrow` would have been sufficient).

In general, each `setting` value makes an assertion about the behavior of the function, without requiring the compiler to prove that this behavior is indeed true. These assertions are made for all world ages. It is thus advisable to limit the use of generic functions that may later be extended to invalidate the assumption (which would cause undefined behavior).

The following `settings` are supported.

- `:consistent`
- `:effect_free`
- `:nothrow`
- `:terminates_globally`
- `:terminates_locally`

- `:notaskstate`
- `:inaccessiblememonly`
- `:noub`
- `:noub_if_noinbounds`
- `:nortcall`
- `:foldable`
- `:removable`
- `:total`

Extended help

`:consistent`

The `:consistent` setting asserts that for egal (===) inputs:

- The manner of termination (return value, exception, non-termination) will always be the same.
- If the method returns, the results will always be egal.

Note

This in particular implies that the method must not return a freshly allocated mutable object. Multiple allocations of mutable objects (even with identical contents) are not egal.

Note

The `:consistent`-cy assertion is made with respect to a particular world range R. More formally, write f for the evaluation of f in world-age i, then this setting requires:

$$iR, jR, x, y : xy \to f(x)f(y)$$

For `@assume_effects`, the range R is `m.primary_world:m.deleted_world` of the annotated or containing method.

For ordinary code instances, R is `ci.min_world:ci.max_world`.

A further implication is that `:consistent` functions may not make their return value dependent on the state of the heap or any other global state that is not constant over the given world age range.

Note

The `:consistent`-cy includes all legal rewrites performed by the optimizer. For example, floating-point fastmath operations are not considered `:consistent`, because the optimizer may rewrite them causing the output to not be `:consistent`, even for the same world age (e.g. because one ran in the interpreter, while the other was optimized).

Note

If `:consistent` functions terminate by throwing an exception, that exception itself is not required to meet the egality requirement specified above.

`:effect_free`

The `:effect_free` setting asserts that the method is free of externally semantically visible side effects. The following is an incomplete list of externally semantically visible side effects:

- Changing the value of a global variable.
- Mutating the heap (e.g. an array or mutable value), except as noted below
- Changing the method table (e.g. through calls to eval)
- File/Network/etc. I/O
- Task switching

However, the following are explicitly not semantically visible, even if they may be observable:

- Memory allocations (both mutable and immutable)
- Elapsed time
- Garbage collection
- Heap mutations of objects whose lifetime does not exceed the method (i.e. were allocated in the method and do not escape).
- The returned value (which is externally visible, but not a side effect)

The rule of thumb here is that an externally visible side effect is anything that would affect the execution of the remainder of the program if the function were not executed.

Note

The `:effect_free` assertion is made both for the method itself and any code that is executed by the method. Keep in mind that the assertion must be valid for all world ages and limit use of this assertion accordingly.

`:nothrow`

The `:nothrow` settings asserts that this method does not throw an exception (i.e. will either always return a value or never return).

Note

It is permissible for `:nothrow` annotated methods to make use of exception handling internally as long as the exception is not rethrown out of the method itself.

Note

If the execution of a method may raise `MethodErrors` and similar exceptions, then the method is not considered as `:nothrow`. However, note that environment-dependent errors like `StackOverflowError` or `InterruptException` are not modeled by this effect and thus a method that may result in `StackOverflowError` does not necessarily need to be `!:nothrow` (although it should usually be `!:terminates` too).

`:terminates_globally`

The `:terminates_globally` settings asserts that this method will eventually terminate (either normally or abnormally), i.e. does not loop indefinitely.

Note

This `:terminates_globally` assertion covers any other methods called by the annotated method.

Note

The compiler will consider this a strong indication that the method will terminate relatively *quickly* and may (if otherwise legal) call this method at compile time. I.e. it is a bad idea to annotate this setting on a method that *technically*, but not *practically*, terminates.

`:terminates_locally`

The `:terminates_locally` setting is like `:terminates_globally`, except that it only applies to syntactic control flow *within* the annotated method. It is thus a much weaker (and thus safer) assertion that allows for the possibility of non-termination if the method calls some other method that does not terminate.

Note

`:terminates_globally` implies `:terminates_locally`.

`:notaskstate`

The `:notaskstate` setting asserts that the method does not use or modify the local task state (task local storage, RNG state, etc.) and may thus be safely moved between tasks without observable results.

Note

The implementation of exception handling makes use of state stored in the task object. However, this state is currently not considered to be within the scope of `:notaskstate` and is tracked separately using the `:nothrow` effect.

Note

The `:notaskstate` assertion concerns the state of the *currently running task*. If a reference to a `Task` object is obtained by some other means that does not consider which task is *currently* running, the `:notaskstate` effect need not be tainted. This is true, even if said task object happens to be `===` to the currently running task.

Note

Access to task state usually also results in the tainting of other effects, such as `:effect_free` (if task state is modified) or `:consistent` (if task state is used in the computation of the result). In particular, code that is not `:notaskstate`, but is `:effect_free` and `:consistent` may still be dead-code-eliminated and thus promoted to `:total`.

`:inaccessiblememonly`

The `:inaccessiblememonly` setting asserts that the method does not access or modify externally accessible mutable memory. This means the method can access or modify mutable memory for newly allocated objects that is not accessible by other methods or top-level execution before return from the method, but it can not access or modify any mutable global state or mutable memory pointed to by its arguments.

Note

Below is an incomplete list of examples that invalidate this assumption:

- a global reference or `getglobal` call to access a mutable global variable
- a global assignment or `setglobal!` call to perform assignment to a non-constant global variable
- `setfield!` call that changes a field of a global mutable variable

Note

This `:inaccessiblememonly` assertion covers any other methods called by the annotated method.

`:noub`

The `:noub` setting asserts that the method will not execute any undefined behavior (for any input). Note that undefined behavior may technically cause the method to violate any other effect assertions (such as `:consistent` or `:effect_free`) as well, but we do not model this, and they assume the absence of undefined behavior.

`:nortcall`

The `:nortcall` setting asserts that the method does not call `Core.Compiler.return_type`, and that any other methods this method might call also do not call `Core.Compiler.return_type`.

Note

To be precise, this assertion can be used when a call to `Core.Compiler.return_type` is not made at runtime; that is, when the result of `Core.Compiler.return_type` is known exactly at compile time and the call is eliminated by the optimizer. However, since whether the result of `Core.Compiler.return_type` is folded at compile time depends heavily on the compiler's implementation, it is generally risky to assert this if the method in question uses `Core.Compiler.return_type` in any form.

`:foldable`

This setting is a convenient shortcut for the set of effects that the compiler requires to be guaranteed to constant fold a call at compile time. It is currently equivalent to the following `settings`:

- `:consistent`
- `:effect_free`
- `:terminates_globally`
- `:noub`
- `:nortcall`

Note

This list in particular does not include `:nothrow`. The compiler will still attempt constant propagation and note any thrown error at compile time. Note however, that by the `:consistent`-cy requirements, any such annotated call must consistently throw given the same argument values.

Note

An explicit `@inbounds` annotation inside the function will also disable constant folding and not be overridden by `:foldable`.

`:removable`

This setting is a convenient shortcut for the set of effects that the compiler requires to be guaranteed to delete a call whose result is unused at compile time. It is currently equivalent to the following `settings`:

- `:effect_free`
- `:nothrow`
- `:terminates_globally`

`:total`

This `setting` is the maximum possible set of effects. It currently implies the following other `settings`:

- `:consistent`
- `:effect_free`

- :nothrow
- :terminates_globally
- :notaskstate
- :inaccessiblememonly
- :noub
- :nortcall

Warning

:total is a very strong assertion and will likely gain additional semantics in future versions of Julia (e.g. if additional effects are added and included in the definition of :total). As a result, it should be used with care. Whenever possible, prefer to use the minimum possible set of specific effect assertions required for a particular application. In cases where a large number of effect overrides apply to a set of functions, a custom macro is recommended over the use of :total.

Negated effects

Effect names may be prefixed by ! to indicate that the effect should be removed from an earlier meta effect. For example, :total !:nothrow indicates that while the call is generally total, it may however throw.

source

44.11 Managing deprecations

Base.@deprecate – Macro.

```
@deprecate old new [export_old=true]
```

Deprecate method old and specify the replacement call new, defining a new method old with the specified signature in the process.

To prevent old from being exported, set export_old to false.

See also Base.depwarn().

Julia 1.5

As of Julia 1.5, functions defined by @deprecate do not print warning when julia is run without the --depwarn=yes flag set, as the default value of --depwarn option is no. The warnings are printed from tests run by Pkg.test().

Examples

```
julia> @deprecate old_export(x) new(x)
old_export (generic function with 1 method)

julia> @deprecate old_public(x) new(x) false
old_public (generic function with 1 method)
```

Calls to `@deprecate` without explicit type-annotations will define deprecated methods accepting any number of positional and keyword arguments of type `Any`.

Julia 1.9

Keyword arguments are forwarded when there is no explicit type annotation as of Julia 1.9. For older versions, you can manually forward positional and keyword arguments by doing `@deprecate old(args...; kwargs...) new(args...; kwargs...)`.

To restrict deprecation to a specific signature, annotate the arguments of `old`. For example,

```
julia> new(x::Int) = x;

julia> new(x::Float64) = 2x;

julia> @deprecate old(x::Int) new(x);

julia> methods(old)
# 1 method for generic function "old" from Main:
 [1] old(x::Int64)
     @ deprecated.jl:94
```

will define and deprecate a method `old(x::Int)` that mirrors `new(x::Int)` but will not define nor deprecate the method `old(x::Float64)`.

source

`Base.depwarn` – Function.

```
Base.depwarn(msg::String, funcsym::Symbol; force=false)
```

Print `msg` as a deprecation warning. The symbol `funcsym` should be the name of the calling function, which is used to ensure that the deprecation warning is only printed the first time for each call place. Set `force=true` to force the warning to always be shown, even if Julia was started with `--depwarn=no` (the default).

See also `@deprecate`.

Examples

```
function deprecated_func()
    Base.depwarn("Don't use `deprecated_func()`!", :deprecated_func)

    1 + 1
end
```

source

44.12 Missing Values

`Base.Missing` – Type.

```
Missing
```

A type with no fields whose singleton instance `missing` is used to represent missing values.

See also: `skipmissing`, `nonmissingtype`, `Nothing`.

source

`Base.missing` – Constant.

```
missing
```

The singleton instance of type `Missing` representing a missing value.

See also: `NaN`, `skipmissing`, `nonmissingtype`.

source

`Base.coalesce` – Function.

```
coalesce(x...)
```

Return the first value in the arguments which is not equal to `missing`, if any. Otherwise return `missing`.

See also `skipmissing`, `something`, `@coalesce`.

Examples

```
julia> coalesce(missing, 1)
1

julia> coalesce(1, missing)
1

julia> coalesce(nothing, 1)  # returns `nothing`

julia> coalesce(missing, missing)
missing
```

source

`Base.@coalesce` – Macro.

```
@coalesce(x...)
```

Short-circuiting version of `coalesce`.

Examples

```
julia> f(x) = (println("f($x)"); missing);

julia> a = 1;

julia> a = @coalesce a f(2) f(3) error("`a` is still missing")
1

julia> b = missing;

julia> b = @coalesce b f(2) f(3) error("`b` is still missing")
```

```
f(2)
f(3)
ERROR: `b` is still missing
[...]
```

Julia 1.7

This macro is available as of Julia 1.7.

source

Base.ismissing – Function.

```
ismissing(x)
```

Indicate whether x is missing.

See also: skipmissing, isnothing, isnan.

source

Base.skipmissing – Function.

```
skipmissing(itr)
```

Return an iterator over the elements in itr skipping missing values. The returned object can be indexed using indices of itr if the latter is indexable. Indices corresponding to missing values are not valid: they are skipped by keys and eachindex, and a MissingException is thrown when trying to use them.

Use collect to obtain an Array containing the non-missing values in itr. Note that even if itr is a multidimensional array, the result will always be a Vector since it is not possible to remove missings while preserving dimensions of the input.

See also coalesce, ismissing, something.

Examples

```
julia> x = skipmissing([1, missing, 2])
skipmissing(Union{Missing, Int64}[1, missing, 2])

julia> sum(x)
3

julia> x[1]
1

julia> x[2]
ERROR: MissingException: the value at index (2,) is missing
[...]

julia> argmax(x)
3

julia> collect(keys(x))
```

```
2-element Vector{Int64}:
 1
 3

julia> collect(skipmissing([1, missing, 2]))
2-element Vector{Int64}:
 1
 2

julia> collect(skipmissing([1 missing; 2 missing]))
2-element Vector{Int64}:
 1
 2
```

source

Base.nonmissingtype – Function.

```
nonmissingtype(T::Type)
```

If T is a union of types containing Missing, return a new type with Missing removed.

Examples

```
julia> nonmissingtype(Union{Int64,Missing})
Int64

julia> nonmissingtype(Any)
Any
```

Julia 1.3

This function is exported as of Julia 1.3.

source

44.13 System

Base.run – Function.

```
run(command, args...; wait::Bool = true)
```

Run a command object, constructed with backticks (see the Running External Programs section in the manual). Throws an error if anything goes wrong, including the process exiting with a non-zero status (when wait is true).

The args... allow you to pass through file descriptors to the command, and are ordered like regular unix file descriptors (eg stdin, stdout, stderr, FD(3), FD(4)...).

If wait is false, the process runs asynchronously. You can later wait for it and check its exit status by calling success on the returned process object. If the command spawns only a single process, a Process object is returned and the exit code can be retrieved via the exitcode field; see wait for more details.

When `wait` is false, the process' I/O streams are directed to `devnull`. When `wait` is true, I/O streams are shared with the parent process. Use `pipeline` to control I/O redirection.

See also: `Cmd`.

source

`Base.devnull` – Constant.

```
devnull
```

Used in a stream redirect to discard all data written to it. Essentially equivalent to `/dev/null` on Unix or `NUL` on Windows. Usage:

```
run(pipeline(`cat test.txt`, devnull))
```

source

`Base.success` – Function.

```
success(command)
```

Run a command object, constructed with backticks (see the Running External Programs section in the manual), and tell whether it was successful (exited with a code of 0). An exception is raised if the process cannot be started.

source

`Base.process_running` – Function.

```
process_running(p::Process)
```

Determine whether a process is currently running.

source

`Base.process_exited` – Function.

```
process_exited(p::Process)
```

Determine whether a process has exited.

source

`Base.kill` – Method.

```
kill(p::Process, signum=Base.SIGTERM)
```

Send a signal to a process. The default is to terminate the process. Returns successfully if the process has already exited, but throws an error if killing the process failed for other reasons (e.g. insufficient permissions).

source

`Base.Sys.set_process_title` – Function.

```
Sys.set_process_title(title::AbstractString)
```

Set the process title. No-op on some operating systems.

source

Base.Sys.get_process_title – Function.

```
Sys.get_process_title()
```

Get the process title. On some systems, will always return an empty string.

source

Base.ignorestatus – Function.

```
ignorestatus(command)
```

Mark a command object so that running it will not throw an error if the result code is non-zero.

source

Base.detach – Function.

```
detach(command)
```

Mark a command object so that it will be run in a new process group, allowing it to outlive the julia process, and not have Ctrl-C interrupts passed to it.

source

Base.Cmd – Type.

```
Cmd(cmd::Cmd; ignorestatus, detach, windows_verbatim, windows_hide, env, dir, uid, gid)
Cmd(exec::Vector{String})
```

Construct a new Cmd object, representing an external program and arguments, from cmd, while changing the settings of the optional keyword arguments:

- ignorestatus::Bool: If true (defaults to false), then the Cmd will not throw an error if the return code is nonzero.
- detach::Bool: If true (defaults to false), then the Cmd will be run in a new process group, allowing it to outlive the julia process and not have Ctrl-C passed to it.
- windows_verbatim::Bool: If true (defaults to false), then on Windows the Cmd will send a command-line string to the process with no quoting or escaping of arguments, even arguments containing spaces. (On Windows, arguments are sent to a program as a single "command-line" string, and programs are responsible for parsing it into arguments. By default, empty arguments and arguments with spaces or tabs are quoted with double quotes " in the command line, and \ or " are preceded by backslashes. windows_verbatim=true is useful for launching programs that parse their command line in nonstandard ways.) Has no effect on non-Windows systems.
- windows_hide::Bool: If true (defaults to false), then on Windows no new console window is displayed when the Cmd is executed. This has no effect if a console is already open or on non-Windows systems.

- env: Set environment variables to use when running the Cmd. env is either a dictionary mapping strings to strings, an array of strings of the form "var=val", an array or tuple of "var"=>val pairs. In order to modify (rather than replace) the existing environment, initialize env with copy(ENV) and then set env["var"]=val as desired. To add to an environment block within a Cmd object without replacing all elements, use addenv() which will return a Cmd object with the updated environment.
- dir::AbstractString: Specify a working directory for the command (instead of the current directory).
- uid::Union{Nothing,UInt32}: Set the user ID for the process (Unix only).
- gid::Union{Nothing,UInt32}: Set the group ID for the process (Unix only).

For any keywords that are not specified, the current settings from cmd are used.

Note that the Cmd(exec) constructor does not create a copy of exec. Any subsequent changes to exec will be reflected in the Cmd object.

The most common way to construct a Cmd object is with command literals (backticks), e.g.

```
`ls -l`
```

This can then be passed to the Cmd constructor to modify its settings, e.g.

```
Cmd(`echo "Hello world"`, ignorestatus=true, detach=false)
```

source

Base.setenv – Function.

```
setenv(command::Cmd, env; dir)
```

Set environment variables to use when running the given command. env is either a dictionary mapping strings to strings, an array of strings of the form "var=val", or zero or more "var"=>val pair arguments. In order to modify (rather than replace) the existing environment, create env through copy(ENV) and then setting env["var"]=val as desired, or use addenv.

The dir keyword argument can be used to specify a working directory for the command. dir defaults to the currently set dir for command (which is the current working directory if not specified already).

See also Cmd, addenv, ENV, pwd.

source

Base.addenv – Function.

```
addenv(command::Cmd, env...; inherit::Bool = true)
```

Merge new environment mappings into the given Cmd object, returning a new Cmd object. Duplicate keys are replaced. If command does not contain any environment values set already, it inherits the current environment at time of addenv() call if inherit is true. Keys with value nothing are deleted from the env.

See also Cmd, setenv, ENV.

Julia 1.6

This function requires Julia 1.6 or later.

source

Base.withenv – Function.

```
withenv(f, kv::Pair...)
```

Execute `f` in an environment that is temporarily modified (not replaced as in `setenv`) by zero or more `"var"=>val` arguments `kv`. `withenv` is generally used via the `withenv(kv...) do ... end` syntax. A value of `nothing` can be used to temporarily unset an environment variable (if it is set). When `withenv` returns, the original environment has been restored.

Warning

Changing the environment is not thread-safe. For running external commands with a different environment from the parent process, prefer using addenv over `withenv`.

source

Base.shell_escape – Function.

```
shell_escape(args::Union{Cmd,AbstractString...}; special::AbstractString="")
```

The unexported `shell_escape` function is the inverse of the unexported `Base.shell_split()` function: it takes a string or command object and escapes any special characters in such a way that calling `Base.shell_split()` on it would give back the array of words in the original command. The `special` keyword argument controls what characters in addition to whitespace, backslashes, quotes and dollar signs are considered to be special (default: none).

Examples

```
julia> Base.shell_escape("cat", "/foo/bar baz", "&&", "echo", "done")
"cat '/foo/bar baz' && echo done"

julia> Base.shell_escape("echo", "this", "&&", "that")
"echo this && that"
```

source

Base.shell_split – Function.

```
shell_split(command::AbstractString)
```

Split a shell command string into its individual components.

Examples

```
julia> Base.shell_split("git commit -m 'Initial commit'")
4-element Vector{String}:
 "git"
 "commit"
 "-m"
 "Initial commit"
```

source

Base.shell_escape_posixly – Function.

```
shell_escape_posixly(args::Union{Cmd,AbstractString...})
```

The unexported shell_escape_posixly function takes a string or command object and escapes any special characters in such a way that it is safe to pass it as an argument to a posix shell.

See also: Base.shell_escape()

Examples

```
julia> Base.shell_escape_posixly("cat", "/foo/bar baz", "&&", "echo", "done")
"cat '/foo/bar baz' '&&' echo done"

julia> Base.shell_escape_posixly("echo", "this", "&&", "that")
"echo this '&&' that"
```

source

Base.shell_escape_csh – Function.

```
shell_escape_csh(args::Union{Cmd,AbstractString...})
shell_escape_csh(io::IO, args::Union{Cmd,AbstractString...})
```

This function quotes any metacharacters in the string arguments such that the string returned can be inserted into a command-line for interpretation by the Unix C shell (csh, tcsh), where each string argument will form one word.

In contrast to a POSIX shell, csh does not support the use of the backslash as a general escape character in double-quoted strings. Therefore, this function wraps strings that might contain metacharacters in single quotes, except for parts that contain single quotes, which it wraps in double quotes instead. It switches between these types of quotes as needed. Linefeed characters are escaped with a backslash.

This function should also work for a POSIX shell, except if the input string contains a linefeed ("\n") character.

See also: Base.shell_escape_posixly()

source

Base.shell_escape_wincmd – Function.

```
shell_escape_wincmd(s::AbstractString)
shell_escape_wincmd(io::IO, s::AbstractString)
```

The unexported shell_escape_wincmd function escapes Windows cmd.exe shell meta characters. It escapes ()!^<>&| by placing a ^ in front. An @ is only escaped at the start of the string. Pairs of " characters and the strings they enclose are passed through unescaped. Any remaining " is escaped with ^ to ensure that the number of unescaped " characters in the result remains even.

Since cmd.exe substitutes variable references (like %USER%) *before* processing the escape characters ^ and ", this function makes no attempt to escape the percent sign (%), the presence of % in the input may cause severe breakage, depending on where the result is used.

Input strings with ASCII control characters that cannot be escaped (NUL, CR, LF) will cause an `ArgumentError` exception.

The result is safe to pass as an argument to a command call being processed by `CMD.exe /S /C " ... "` (with surrounding double-quote pair) and will be received verbatim by the target application if the input does not contain `%` (else this function will fail with an ArgumentError). The presence of `%` in the input string may result in command injection vulnerabilities and may invalidate any claim of suitability of the output of this function for use as an argument to cmd (due to the ordering described above), so use caution when assembling a string from various sources.

This function may be useful in concert with the `windows_verbatim` flag to `Cmd` when constructing process pipelines.

```
wincmd(c::String) =
   run(Cmd(Cmd(["cmd.exe", "/s /c \" $c \""]);
           windows_verbatim=true))
wincmd_echo(s::String) =
   wincmd("echo " * Base.shell_escape_wincmd(s))
wincmd_echo("hello $(ENV["USERNAME"]) & the \"whole\" world! (=^I^=)")
```

But take note that if the input string `s` contains a `%`, the argument list and echo'ed text may get corrupted, resulting in arbitrary command execution. The argument can alternatively be passed as an environment variable, which avoids the problem with `%` and the need for the `windows_verbatim` flag:

```
cmdargs = Base.shell_escape_wincmd("Passing args with %cmdargs% works 100%!")
run(setenv(`cmd /C echo %cmdargs%`, "cmdargs" => cmdargs))
```

Warning

The argument parsing done by CMD when calling batch files (either inside `.bat` files or as arguments to them) is not fully compatible with the output of this function. In particular, the processing of `%` is different.

Important

Due to a peculiar behavior of the CMD parser/interpreter, each command after a literal `|` character (indicating a command pipeline) must have `shell_escape_wincmd` applied twice since it will be parsed twice by CMD. This implies ENV variables would also be expanded twice! For example:

```
to_print = "All for 1 & 1 for all!"
to_print_esc = Base.shell_escape_wincmd(Base.shell_escape_wincmd(to_print))
run(Cmd(Cmd(["cmd", "/S /C \" break | echo $(to_print_esc) \""]),
↪  windows_verbatim=true))
```

With an I/O stream parameter `io`, the result will be written there, rather than returned as a string.

See also `Base.escape_microsoft_c_args()`, `Base.shell_escape_posixly()`.

Examples

```
julia> Base.shell_escape_wincmd("a^\"^o\"^u\"")
"a^^\"^o\"^^u^\""
```

source

Base.escape_microsoft_c_args – Function.

```
escape_microsoft_c_args(args::Union{Cmd,AbstractString...})
escape_microsoft_c_args(io::IO, args::Union{Cmd,AbstractString...})
```

Convert a collection of string arguments into a string that can be passed to many Windows command-line applications.

Microsoft Windows passes the entire command line as a single string to the application (unlike POSIX systems, where the shell splits the command line into a list of arguments). Many Windows API applications (including julia.exe), use the conventions of the Microsoft C/C++ runtime to split that command line into a list of strings.

This function implements an inverse for a parser compatible with these rules. It joins command-line arguments to be passed to a Windows C/C++/Julia application into a command line, escaping or quoting the meta characters space, TAB, double quote and backslash where needed.

See also Base.shell_escape_wincmd(), Base.escape_raw_string().

source

Base.setcpuaffinity – Function.

```
setcpuaffinity(original_command::Cmd, cpus) -> command::Cmd
```

Set the CPU affinity of the command by a list of CPU IDs (1-based) cpus. Passing cpus = nothing means to unset the CPU affinity if the original_command has any.

This function is supported only in Linux and Windows. It is not supported in macOS because libuv does not support affinity setting.

Julia 1.8

This function requires at least Julia 1.8.

Examples

In Linux, the taskset command line program can be used to see how setcpuaffinity works.

```
julia> run(setcpuaffinity(`sh -c 'taskset -p $$'`, [1, 2, 5]));
pid 2273's current affinity mask: 13
```

Note that the mask value 13 reflects that the first, second, and the fifth bits (counting from the least significant position) are turned on:

```
julia> 0b010011
0x13
```

source

Base.pipeline – Method.

```
pipeline(from, to, ...)
```

Create a pipeline from a data source to a destination. The source and destination can be commands, I/O streams, strings, or results of other `pipeline` calls. At least one argument must be a command. Strings refer to filenames. When called with more than two arguments, they are chained together from left to right. For example, `pipeline(a,b,c)` is equivalent to `pipeline(pipeline(a,b),c)`. This provides a more concise way to specify multi-stage pipelines.

Examples:

```
run(pipeline(`ls`, `grep xyz`))
run(pipeline(`ls`, "out.txt"))
run(pipeline("out.txt", `grep xyz`))
```

source

`Base.pipeline` – Method.

```
pipeline(command; stdin, stdout, stderr, append=false)
```

Redirect I/O to or from the given `command`. Keyword arguments specify which of the command's streams should be redirected. `append` controls whether file output appends to the file. This is a more general version of the 2-argument `pipeline` function. `pipeline(from, to)` is equivalent to `pipeline(from, stdout=to)` when `from` is a command, and to `pipeline(to, stdin=from)` when `from` is another kind of data source.

Examples:

```
run(pipeline(`dothings`, stdout="out.txt", stderr="errs.txt"))
run(pipeline(`update`, stdout="log.txt", append=true))
```

source

`Base.Libc.gethostname` – Function.

```
gethostname()::String
```

Get the local machine's host name.

source

`Base.Libc.getpid` – Function.

```
getpid()::Int32
```

Get Julia's process ID.

source

```
getpid(process)::Int32
```

Get the child process ID, if it still exists.

Julia 1.1

This function requires at least Julia 1.1.

source

Base.Libc.time – Method.

```
time()::Float64
```

Get the system time in seconds since the epoch, with fairly high (typically, microsecond) resolution.

See also time_ns.

source

Base.time_ns – Function.

```
time_ns()::UInt64
```

Get the time in nanoseconds relative to some machine-specific arbitrary time in the past. The primary use is for measuring elapsed times during program execution. The return value is guaranteed to be monotonic (mod 2^{64}) while the system is running, and is unaffected by clock drift or changes to local calendar time, but it may change arbitrarily across system reboots or suspensions.

(Although the returned time is always in nanoseconds, the timing resolution is platform-dependent.)

source

Base.@time – Macro.

```
@time expr
@time "description" expr
```

A macro to execute an expression, printing the time it took to execute, the number of allocations, and the total number of bytes its execution caused to be allocated, before returning the value of the expression. Any time spent garbage collecting (gc), compiling new code, or recompiling invalidated code is shown as a percentage. Any lock conflicts where a ReentrantLock had to wait are shown as a count.

Optionally provide a description string to print before the time report.

In some cases the system will look inside the @time expression and compile some of the called code before execution of the top-level expression begins. When that happens, some compilation time will not be counted. To include this time you can run @time @eval ....

See also @showtime, @timev, @timed, @elapsed, @allocated, and @allocations.

Note

For more serious benchmarking, consider the @btime macro from the BenchmarkTools.jl package which among other things evaluates the function multiple times in order to reduce noise.

Julia 1.8

The option to add a description was introduced in Julia 1.8.

Recompilation time being shown separately from compilation time was introduced in Julia 1.8

Julia 1.11

The reporting of any lock conflicts was added in Julia 1.11.

```
julia> x = rand(10,10);

julia> @time x * x;
  0.606588 seconds (2.19 M allocations: 116.555 MiB, 3.75% gc time, 99.94% compilation time)

julia> @time x * x;
  0.000009 seconds (1 allocation: 896 bytes)

julia> @time begin
           sleep(0.3)
           1+1
       end
  0.301395 seconds (8 allocations: 336 bytes)
2

julia> @time "A one second sleep" sleep(1)
A one second sleep: 1.005750 seconds (5 allocations: 144 bytes)

julia> for loop in 1:3
           @time loop sleep(1)
       end
1: 1.006760 seconds (5 allocations: 144 bytes)
2: 1.001263 seconds (5 allocations: 144 bytes)
3: 1.003676 seconds (5 allocations: 144 bytes)
```

source

Base.@showtime – Macro.

```
@showtime expr
```

Like @time but also prints the expression being evaluated for reference.

Julia 1.8

This macro was added in Julia 1.8.

See also @time.

```
julia> @showtime sleep(1)
sleep(1): 1.002164 seconds (4 allocations: 128 bytes)
```

source

Base.@timev – Macro.

```
@timev expr
@timev "description" expr
```

This is a verbose version of the @time macro. It first prints the same information as @time, then any non-zero memory allocation counters, and then returns the value of the expression.

Optionally provide a description string to print before the time report.

Julia 1.8

The option to add a description was introduced in Julia 1.8.

See also `@time`, `@timed`, `@elapsed`, `@allocated`, and `@allocations`.

```
julia> x = rand(10,10);

julia> @timev x * x;
  0.546770 seconds (2.20 M allocations: 116.632 MiB, 4.23% gc time, 99.94% compilation time)
elapsed time (ns): 546769547
gc time (ns):      23115606
bytes allocated:   122297811
pool allocs:       2197930
non-pool GC allocs:1327
malloc() calls:    36
realloc() calls:   5
GC pauses:         3

julia> @timev x * x;
  0.000010 seconds (1 allocation: 896 bytes)
elapsed time (ns): 9848
bytes allocated:   896
pool allocs:       1
```

source

Base.@timed – Macro.

```
@timed
```

A macro to execute an expression, and return the value of the expression, elapsed time in seconds, total bytes allocated, garbage collection time, an object with various memory allocation counters, compilation time in seconds, and recompilation time in seconds. Any lock conflicts where a `ReentrantLock` had to wait are shown as a count.

In some cases the system will look inside the `@timed` expression and compile some of the called code before execution of the top-level expression begins. When that happens, some compilation time will not be counted. To include this time you can run `@timed @eval ...`.

See also `@time`, `@timev`, `@elapsed`, `@allocated`, `@allocations`, and `@lock_conflicts`.

```
julia> stats = @timed rand(10^6);

julia> stats.time
0.006634834

julia> stats.bytes
8000256

julia> stats.gctime
0.0055765
```

```
julia> propertynames(stats.gcstats)
(:allocd, :malloc, :realloc, :poolalloc, :bigalloc, :freecall, :total_time, :pause,
↪  :full_sweep)

julia> stats.gcstats.total_time
5576500

julia> stats.compile_time
0.0

julia> stats.recompile_time
0.0
```

Julia 1.5

The return type of this macro was changed from `Tuple` to `NamedTuple` in Julia 1.5.

Julia 1.11

The `lock_conflicts`, `compile_time`, and `recompile_time` fields were added in Julia 1.11.

source

`Base.@elapsed` – Macro.

```
@elapsed
```

A macro to evaluate an expression, discarding the resulting value, instead returning the number of seconds it took to execute as a floating-point number.

In some cases the system will look inside the `@elapsed` expression and compile some of the called code before execution of the top-level expression begins. When that happens, some compilation time will not be counted. To include this time you can run `@elapsed @eval ...`.

See also `@time`, `@timev`, `@timed`, `@allocated`, and `@allocations`.

```
julia> @elapsed sleep(0.3)
0.301391426
```

source

`Base.@allocated` – Macro.

```
@allocated
```

A macro to evaluate an expression, discarding the resulting value, instead returning the total number of bytes allocated during evaluation of the expression.

If the expression is a function call, an effort is made to measure only allocations from the argument expressions and during the function, excluding any overhead from calling it and not performing constant propagation with the provided argument values. If you want to include those effects, i.e. measuring the call site as well, use the syntax `@allocated (()->f(1))()`.

It is recommended to measure function calls with only simple argument expressions, e.g. `x = []; @allocated f(x)` instead of `@allocated f([])` to clarify that only `f` is being measured.

For more complex expressions, the code is simply run in place and therefore may see allocations due to the surrounding context. For example it is possible for `@allocated f(1)` and `@allocated x = f(1)` to give different results.

See also `@allocations`, `@time`, `@timev`, `@timed`, and `@elapsed`.

```
julia> @allocated rand(10^6)
8000080
```

source

`Base.@allocations` – Macro.

```
@allocations
```

A macro to evaluate an expression, discard the resulting value, and instead return the total number of allocations during evaluation of the expression.

See also `@allocated`, `@time`, `@timev`, `@timed`, and `@elapsed`.

```
julia> @allocations rand(10^6)
2
```

Julia 1.9

This macro was added in Julia 1.9.

source

`Base.@lock_conflicts` – Macro.

```
@lock_conflicts
```

A macro to evaluate an expression, discard the resulting value, and instead return the total number of lock conflicts during evaluation, where a lock attempt on a `ReentrantLock` resulted in a wait because the lock was already held.

See also `@time`, `@timev` and `@timed`.

```
julia> @lock_conflicts begin
    l = ReentrantLock()
    Threads.@threads for i in 1:Threads.nthreads()
        lock(l) do
        sleep(1)
        end
    end
end
5
```

Julia 1.11

This macro was added in Julia 1.11.

source

Base.TRACE_EVAL – Constant.

```
Base.TRACE_EVAL
```

Global control for expression tracing during top-level evaluation. This setting takes priority over the `--trace-eval` command-line option.

Set to:

- `nothing` - use the command-line `--trace-eval` setting (default)
- `:no` - disable expression tracing
- `:loc` - show only location information during evaluation
- `:full` - show full expressions being evaluated

Examples

```
# Enable full expression tracing
Base.TRACE_EVAL = :full

# Show only locations
Base.TRACE_EVAL = :loc

# Disable tracing (overrides command-line setting)
Base.TRACE_EVAL = :no

# Reset to use command-line setting
Base.TRACE_EVAL = nothing
```

See also: Command-line Interface for the `--trace-eval` option.

source

Base.EnvDict – Type.

```
EnvDict()::EnvDict
```

A singleton of this type provides a hash table interface to environment variables.

source

Base.ENV – Constant.

```
ENV
```

Reference to the singleton `EnvDict`, providing a dictionary interface to system environment variables.

(On Windows, system environment variables are case-insensitive, and `ENV` correspondingly converts all keys to uppercase for display, iteration, and copying. Portable code should not rely on the ability

to distinguish variables by case, and should beware that setting an ostensibly lowercase variable may result in an uppercase ENV key.)

Warning

Mutating the environment is not thread-safe.

Examples

```
julia> ENV
Base.EnvDict with "50" entries:
  "SECURITYSESSIONID"             => "123"
  "USER"                          => "username"
  "MallocNanoZone"                => "0"
  □                               => □

julia> ENV["JULIA_EDITOR"] = "vim"
"vim"

julia> ENV["JULIA_EDITOR"]
"vim"
```

See also: withenv, addenv.

source

Base.Sys.STDLIB – Constant.

```
Sys.STDLIB::String
```

A string containing the full path to the directory containing the stdlib packages.

source

Base.Sys.isunix – Function.

```
Sys.isunix([os])
```

Predicate for testing if the OS provides a Unix-like interface. See documentation in Handling Operating System Variation.

source

Base.Sys.isapple – Function.

```
Sys.isapple([os])
```

Predicate for testing if the OS is a derivative of Apple Macintosh OS X or Darwin. See documentation in Handling Operating System Variation.

source

Base.Sys.islinux – Function.

```
Sys.islinux([os])
```

Predicate for testing if the OS is a derivative of Linux. See documentation in Handling Operating System Variation.

source

Base.Sys.isbsd – Function.

```
Sys.isbsd([os])
```

Predicate for testing if the OS is a derivative of BSD. See documentation in Handling Operating System Variation.

Note

The Darwin kernel descends from BSD, which means that `Sys.isbsd()` is `true` on macOS systems. To exclude macOS from a predicate, use `Sys.isbsd() && !Sys.isapple()`.

source

Base.Sys.isfreebsd – Function.

```
Sys.isfreebsd([os])
```

Predicate for testing if the OS is a derivative of FreeBSD. See documentation in Handling Operating System Variation.

Note

Not to be confused with `Sys.isbsd()`, which is `true` on FreeBSD but also on other BSD-based systems. `Sys.isfreebsd()` refers only to FreeBSD.

Julia 1.1

This function requires at least Julia 1.1.

source

Base.Sys.isopenbsd – Function.

```
Sys.isopenbsd([os])
```

Predicate for testing if the OS is a derivative of OpenBSD. See documentation in Handling Operating System Variation.

Note

Not to be confused with `Sys.isbsd()`, which is `true` on OpenBSD but also on other BSD-based systems. `Sys.isopenbsd()` refers only to OpenBSD.

Julia 1.1

This function requires at least Julia 1.1.

source

`Base.Sys.isnetbsd` – Function.

```
Sys.isnetbsd([os])
```

Predicate for testing if the OS is a derivative of NetBSD. See documentation in Handling Operating System Variation.

Note

Not to be confused with `Sys.isbsd()`, which is `true` on NetBSD but also on other BSD-based systems. `Sys.isnetbsd()` refers only to NetBSD.

Julia 1.1

This function requires at least Julia 1.1.

source

`Base.Sys.isdragonfly` – Function.

```
Sys.isdragonfly([os])
```

Predicate for testing if the OS is a derivative of DragonFly BSD. See documentation in Handling Operating System Variation.

Note

Not to be confused with `Sys.isbsd()`, which is `true` on DragonFly but also on other BSD-based systems. `Sys.isdragonfly()` refers only to DragonFly.

Julia 1.1

This function requires at least Julia 1.1.

source

`Base.Sys.iswindows` – Function.

```
Sys.iswindows([os])
```

Predicate for testing if the OS is a derivative of Microsoft Windows NT. See documentation in Handling Operating System Variation.

source

`Base.Sys.windows_version` – Function.

```
Sys.windows_version()
```

Return the version number for the Windows NT Kernel as a `VersionNumber`, i.e. `v"major.minor.build"`, or `v"0.0.0"` if this is not running on Windows.

source

`Base.Sys.free_memory` – Function.

```
Sys.free_memory()
```

Get the total free memory in RAM in bytes.

source

Base.Sys.total_memory – Function.

```
Sys.total_memory()
```

Get the total memory in RAM (including that which is currently used) in bytes. This amount may be constrained, e.g., by Linux control groups. For the unconstrained amount, see `Sys.total_physical_memory()`.

source

Base.Sys.free_physical_memory – Function.

```
Sys.free_physical_memory()
```

Get the free memory of the system in bytes. The entire amount may not be available to the current process; use `Sys.free_memory()` for the actually available amount.

source

Base.Sys.total_physical_memory – Function.

```
Sys.total_physical_memory()
```

Get the total memory in RAM (including that which is currently used) in bytes. The entire amount may not be available to the current process; see `Sys.total_memory()`.

source

Base.Sys.uptime – Function.

```
Sys.uptime()
```

Gets the current system uptime in seconds.

source

Base.Sys.sysimage_target – Function.

```
Sys.sysimage_target()
```

Return the CPU target string that was used to build the current system image.

This function returns the original CPU target specification that was passed to Julia when the system image was compiled. This can be useful for reproducing the same system image or for understanding what CPU features were enabled during compilation.

If the system image was built with the default settings this will return `"native"`.

See also JULIA_CPU_TARGET.

source

Base.Sys.isjsvm – Function.

```
Sys.isjsvm([os])
```

Predicate for testing if Julia is running in a JavaScript VM (JSVM), including e.g. a WebAssembly JavaScript embedding in a web browser.

Julia 1.2

This function requires at least Julia 1.2.

source

Base.Sys.loadavg – Function.

```
Sys.loadavg()
```

Get the load average. See: https://en.wikipedia.org/wiki/Load_(computing).

source

Base.Sys.isexecutable – Function.

```
isexecutable(path::String)
```

Return `true` if the given `path` has executable permissions.

Note

This permission may change before the user executes `path`, so it is recommended to execute the file and handle the error if that fails, rather than calling `isexecutable` first.

Note

Prior to Julia 1.6, this did not correctly interrogate filesystem ACLs on Windows, therefore it would return `true` for any file. From Julia 1.6 on, it correctly determines whether the file is marked as executable or not.

See also `ispath`, `isreadable`, `iswritable`.

source

Base.Sys.isreadable – Function.

```
isreadable(io)::Bool
```

Return `false` if the specified IO object is not readable.

Examples

```
julia> open("myfile.txt", "w") do io
           print(io, "Hello world!");
           isreadable(io)
       end
false

julia> open("myfile.txt", "r") do io
```

```
            isreadable(io)
        end
true

julia> rm("myfile.txt")
```

source

```
isreadable(path::String)
```

Return `true` if the access permissions for the given `path` permitted reading by the current user.

Note

This permission may change before the user calls open, so it is recommended to just call open alone and handle the error if that fails, rather than calling `isreadable` first.

Note

Currently this function does not correctly interrogate filesystem ACLs on Windows, therefore it can return wrong results.

Julia 1.11

This function requires at least Julia 1.11.

See also `ispath`, `isexecutable`, `iswritable`.

source

`Base.Sys.iswritable` – Function.

```
iswritable(io)::Bool
```

Return `false` if the specified IO object is not writable.

Examples

```
julia> open("myfile.txt", "w") do io
            print(io, "Hello world!");
            iswritable(io)
        end
true

julia> open("myfile.txt", "r") do io
            iswritable(io)
        end
false

julia> rm("myfile.txt")
```

source

```
iswritable(path::String)
```

Return `true` if the access permissions for the given `path` permitted writing by the current user.

Note

This permission may change before the user calls open, so it is recommended to just call open alone and handle the error if that fails, rather than calling `iswritable` first.

Note

Currently this function does not correctly interrogate filesystem ACLs on Windows, therefore it can return wrong results.

Julia 1.11

This function requires at least Julia 1.11.

See also `ispath`, `isexecutable`, `isreadable`.

source

`Base.Sys.which` – Function.

```
Sys.which(program_name::String)
```

Given a program name, search the current PATH to find the first binary with the proper executable permissions that can be run and return an absolute path to it, or return `nothing` if no such program is available. If a path with a directory in it is passed in for `program_name`, tests that exact path for executable permissions only (with `.exe` and `.com` extensions added on Windows platforms); no searching of PATH is performed.

source

`Base.Sys.username` – Function.

```
Sys.username()::String
```

Return the username for the current user. If the username cannot be determined or is empty, this function throws an error.

To retrieve a username that is overridable via an environment variable, e.g., USER, consider using

```
user = get(Sys.username, ENV, "USER")
```

Julia 1.11

This function requires at least Julia 1.11.

See also `homedir`.

source

`Base.@static` – Macro.

```
@static
```

Partially evaluate an expression at macro expansion time.

This is useful in cases where a construct would be invalid in some cases, such as a `ccall` to an os-dependent function, or macros defined in packages that are not imported.

`@static` requires a conditional. The conditional can be in an `if` statement, a ternary operator, or `&&||`. The conditional is evaluated by recursively expanding macros, lowering and executing the resulting expressions. Then, the matching branch (if any) is returned. All the other branches of the conditional are deleted before they are macro-expanded (and lowered or executed).

Example

Suppose we want to parse an expression `expr` that is valid only on macOS. We could solve this problem using `@static` with `@static if Sys.isapple() expr end`. In case we had `expr_apple` for macOS and `expr_others` for the other operating systems, the solution with `@static` would be `@static Sys.isapple() ? expr_apple : expr_others`.

source

44.14 Versioning

`Base.VersionNumber` – Type.

```
VersionNumber
```

Version number type which follows the specifications of semantic versioning (semver), composed of major, minor and patch numeric values, followed by pre-release and build alphanumeric annotations.

`VersionNumber` objects can be compared with all of the standard comparison operators (==, <, <=, etc.), with the result following semver v2.0.0-rc.2 rules.

`VersionNumber` has the following public fields:

- `v.major::Integer`
- `v.minor::Integer`
- `v.patch::Integer`
- `v.prerelease::Tuple{Vararg{Union{Integer, AbstractString}}}`
- `v.build::Tuple{Vararg{Union{Integer, AbstractString}}}`

See also `@v_str` to efficiently construct `VersionNumber` objects from semver-format literal strings, `VERSION` for the `VersionNumber` of Julia itself, and Version Number Literals in the manual.

Examples

```
julia> a = VersionNumber(1, 2, 3)
v"1.2.3"

julia> a >= v"1.2"
true

julia> b = VersionNumber("2.0.1-rc1")
```

```
v"2.0.1-rc1"

julia> b >= v"2.0.1"
false
```

source

Base.@v_str – Macro.

```
@v_str
```

String macro used to parse a string to a VersionNumber.

Examples

```
julia> v"1.2.3"
v"1.2.3"

julia> v"2.0.1-rc1"
v"2.0.1-rc1"
```

source

44.15 Errors

Base.error – Function.

```
error(msg...)
```

Raise an ErrorException with a message constructed by string(msg...).

source

```
error(message::AbstractString)
```

Raise an ErrorException with the given message.

source

Core.throw – Function.

```
throw(e)
```

Throw an object as an exception.

See also: rethrow, error.

source

Base.rethrow – Function.

```
rethrow()
```

Rethrow the current exception from within a `catch` block. The rethrown exception will continue propagation as if it had not been caught.

Note

The alternative form `rethrow(e)` allows you to associate an alternative exception object `e` with the current backtrace. However this misrepresents the program state at the time of the error so you're encouraged to instead throw a new exception using `throw(e)`. In Julia 1.1 and above, using `throw(e)` will preserve the root cause exception on the stack, as described in `current_exceptions`.

source

`Base.backtrace` – Function.

```
backtrace()
```

Get a backtrace object for the current program point.

source

`Base.catch_backtrace` – Function.

```
catch_backtrace()
```

Get the backtrace of the current exception, for use within `catch` blocks.

source

`Base.current_exceptions` – Function.

```
current_exceptions(task::Task=current_task(); [backtrace::Bool=true])
```

Get the stack of exceptions currently being handled. For nested catch blocks there may be more than one current exception in which case the most recently thrown exception is last in the stack. The stack is returned as an `ExceptionStack` which is an AbstractVector of named tuples `(exception,backtrace)`. If `backtrace` is false, the backtrace in each pair will be set to `nothing`.

Explicitly passing `task` will return the current exception stack on an arbitrary task. This is useful for inspecting tasks which have failed due to uncaught exceptions.

Julia 1.7

This function went by the experimental name `catch_stack()` in Julia 1.1–1.6, and had a plain Vector-of-tuples as a return type.

source

`Base.@assert` – Macro.

```
@assert cond [text]
```

Throw an `AssertionError` if `cond` is `false`. This is the preferred syntax for writing assertions, which are conditions that are assumed to be true, but that the user might decide to check anyways, as an aid to debugging if they fail. The optional message `text` is displayed upon assertion failure.

Warning

An assert might be disabled at some optimization levels. Assert should therefore only be used as a debugging tool and not used for authentication verification (e.g., verifying passwords or checking array bounds). The code must not rely on the side effects of running cond for the correct behavior of a function.

Examples

```
julia> @assert iseven(3) "3 is an odd number!"
ERROR: AssertionError: 3 is an odd number!

julia> @assert isodd(3) "What even are numbers?"
```

source

`Base.Experimental.register_error_hint` – Function.

```
Experimental.register_error_hint(handler, exceptiontype)
```

Register a "hinting" function `handler(io, exception)` that can suggest potential ways for users to circumvent errors. `handler` should examine `exception` to see whether the conditions appropriate for a hint are met, and if so generate output to `io`. Packages should call `register_error_hint` from within their `__init__` function.

For specific exception types, `handler` is required to accept additional arguments:

- `MethodError`: provide `handler(io, exc::MethodError, argtypes, kwargs)`, which splits the combined arguments into positional and keyword arguments.

When issuing a hint, the output should typically start with `\n`.

If you define custom exception types, your `showerror` method can support hints by calling `Experimental.show_error_hints`.

Examples

```
julia> module Hinter

       only_int(x::Int)      = 1
       any_number(x::Number) = 2

       function __init__()
           Base.Experimental.register_error_hint(MethodError) do io, exc, argtypes, kwargs
               if exc.f == only_int
                    # Color is not necessary, this is just to show it's possible.
                    print(io, "\nDid you mean to call ")
                    printstyled(io, "`any_number`?", color=:cyan)
               end
           end
       end

       end
```

Then if you call `Hinter.only_int` on something that isn't an `Int` (thereby triggering a `MethodError`), it issues the hint:

```
julia> Hinter.only_int(1.0)
ERROR: MethodError: no method matching only_int(::Float64)
The function `only_int` exists, but no method is defined for this combination of argument
↪  types.
Did you mean to call `any_number`?
Closest candidates are:
    ...
```

Julia 1.5

Custom error hints are available as of Julia 1.5.

Warning

This interface is experimental and subject to change or removal without notice. To insulate yourself against changes, consider putting any registrations inside an `if isdefined(Base.Experimental, :register_error_hint) ... end` block.

source

`Base.Experimental.show_error_hints` – Function.

```
Experimental.show_error_hints(io, ex, args...)
```

Invoke all handlers from `Experimental.register_error_hint` for the particular exception type `typeof(ex)` and all of its supertypes. `args` must contain any other arguments expected by the handler for that type.

Julia 1.5

Custom error hints are available as of Julia 1.5.

Warning

This interface is experimental and subject to change or removal without notice.

source

`Core.ArgumentError` – Type.

```
ArgumentError(msg)
```

The arguments passed to a function are invalid. `msg` is a descriptive error message.

source

`Core.AssertionError` – Type.

```
AssertionError([msg])
```

The asserted condition did not evaluate to `true`. Optional argument `msg` is a descriptive error string.

Examples

```
julia> @assert false "this is not true"
ERROR: AssertionError: this is not true
```

`AssertionError` is usually thrown from `@assert`.

source

`Core.BoundsError` – Type.

```
BoundsError([a],[i])
```

An indexing operation into an array, `a`, tried to access an out-of-bounds element at index `i`.

Examples

```
julia> A = fill(1.0, 7);

julia> A[8]
ERROR: BoundsError: attempt to access 7-element Vector{Float64} at index [8]


julia> B = fill(1.0, (2,3));

julia> B[2, 4]
ERROR: BoundsError: attempt to access 2×3 Matrix{Float64} at index [2, 4]


julia> B[9]
ERROR: BoundsError: attempt to access 2×3 Matrix{Float64} at index [9]
```

source

`Base.CompositeException` – Type.

```
CompositeException
```

Wrap a `Vector` of exceptions thrown by a `Task` (e.g. generated from a remote worker over a channel or an asynchronously executing local I/O write or a remote worker under `pmap`) with information about the series of exceptions. For example, if a group of workers are executing several tasks, and multiple workers fail, the resulting `CompositeException` will contain a "bundle" of information from each worker indicating where and why the exception(s) occurred.

source

`Base.DimensionMismatch` – Type.

```
DimensionMismatch([msg])
```

The objects called do not have matching dimensionality. Optional argument `msg` is a descriptive error string.

source

`Core.DivideError` – Type.

```
DivideError()
```

Integer division was attempted with a denominator value of 0.

Examples

```
julia> 2/0
Inf

julia> div(2, 0)
ERROR: DivideError: integer division error
Stacktrace:
[...]
```

source

`Core.DomainError` – Type.

```
DomainError(val)
DomainError(val, msg)
```

The argument `val` to a function or constructor is outside the valid domain.

Examples

```
julia> sqrt(-1)
ERROR: DomainError with -1.0:
sqrt was called with a negative real argument but will only return a complex result if called
↪  with a complex argument. Try sqrt(Complex(x)).
Stacktrace:
[...]
```

source

`Base.EOFError` – Type.

```
EOFError()
```

No more data was available to read from a file or stream.

source

`Core.ErrorException` – Type.

```
ErrorException(msg)
```

Generic error type. The error message, in the `.msg` field, may provide more specific details.

Examples

```
julia> ex = ErrorException("I've done a bad thing");

julia> ex.msg
"I've done a bad thing"
```

source

Core.FieldError – Type.

```
FieldError(type::DataType, field::Symbol)
```

An operation tried to access invalid field on an object of type.

Julia 1.12

Prior to Julia 1.12, invalid field access threw an ErrorException

See getfield

Examples

```
julia> struct AB
           a::Float32
           b::Float64
       end

julia> ab = AB(1, 3)
AB(1.0f0, 3.0)

julia> ab.c # field `c` doesn't exist
ERROR: FieldError: type AB has no field `c`, available fields: `a`, `b`
Stacktrace:
[...]
```

source

Core.InexactError – Type.

```
InexactError(name::Symbol, T, val)
```

Cannot exactly convert val to type T in a method of function name.

Examples

```
julia> convert(Float64, 1+2im)
ERROR: InexactError: Float64(1 + 2im)
Stacktrace:
[...]
```

source

Core.InterruptException – Type.

```
InterruptException()
```

The process was stopped by a terminal interrupt (CTRL+C).

Note that, in Julia script started without -i (interactive) option, InterruptException is not thrown by default. Calling Base.exit_on_sigint(false) in the script can recover the behavior of the REPL. Alternatively, a Julia script can be started with

```
julia -e "include(popfirst!(ARGS))" script.jl
```

to let `InterruptException` be thrown by CTRL+C during the execution.

source

`Base.KeyError` – Type.

```
KeyError(key)
```

An indexing operation into an `AbstractDict` (`Dict`) or `Set` like object tried to access or delete a non-existent element.

source

`Core.LoadError` – Type.

```
LoadError(file::AbstractString, line::Int, error)
```

An error occurred while `include`ing, `require`ing, or `using` a file. The error specifics should be available in the `.error` field.

Julia 1.7

LoadErrors are no longer emitted by `@macroexpand`, `@macroexpand1`, and `macroexpand` as of Julia 1.7.

source

`Core.MethodError` – Type.

```
MethodError(f, args)
```

A method with the required type signature does not exist in the given generic function. Alternatively, there is no unique most-specific method.

source

`Base.MissingException` – Type.

```
MissingException(msg)
```

Exception thrown when a `missing` value is encountered in a situation where it is not supported. The error message, in the `msg` field may provide more specific details.

source

`Core.OutOfMemoryError` – Type.

```
OutOfMemoryError()
```

An operation allocated too much memory for either the system or the garbage collector to handle properly.

source

`Core.ReadOnlyMemoryError` – Type.

```
ReadOnlyMemoryError()
```

An operation tried to write to memory that is read-only.

source

`Core.OverflowError` – Type.

```
OverflowError(msg)
```

The result of an expression is too large for the specified type and will cause a wraparound.

source

`Base.ProcessFailedException` – Type.

```
ProcessFailedException
```

Indicates problematic exit status of a process. When running commands or pipelines, this is thrown to indicate a nonzero exit code was returned (i.e. that the invoked process failed).

source

`Base.TaskFailedException` – Type.

```
TaskFailedException
```

This exception is thrown by a `wait(t)` call when task `t` fails. `TaskFailedException` wraps the failed task `t`.

source

`Core.StackOverflowError` – Type.

```
StackOverflowError()
```

The function call grew beyond the size of the call stack. This usually happens when a call recurses infinitely.

source

`Base.SystemError` – Type.

```
SystemError(prefix::AbstractString, [errno::Int32])
```

A system call failed with an error code (in the `errno` global variable).

source

`Core.TypeError` – Type.

```
TypeError(func::Symbol, context::AbstractString, expected::Type, got)
```

A type assertion failure, or calling an intrinsic function with an incorrect argument type.

source

`Core.UndefKeywordError` – Type.

```
UndefKeywordError(var::Symbol)
```

The required keyword argument `var` was not assigned in a function call.

Examples

```
julia> function my_func(;my_arg)
           return my_arg + 1
       end
my_func (generic function with 1 method)

julia> my_func()
ERROR: UndefKeywordError: keyword argument `my_arg` not assigned
Stacktrace:
 [1] my_func() at ./REPL[1]:2
 [2] top-level scope at REPL[2]:1
```

source

`Core.UndefRefError` – Type.

```
UndefRefError()
```

The item or field is not defined for the given object.

Examples

```
julia> struct MyType
           a::Vector{Int}
           MyType() = new()
       end

julia> A = MyType()
MyType(#undef)

julia> A.a
ERROR: UndefRefError: access to undefined reference
Stacktrace:
[...]
```

source

`Core.UndefVarError` – Type.

```
UndefVarError(var::Symbol, [scope])
```

A symbol in the current scope is not defined.

Examples

```
julia> a
ERROR: UndefVarError: `a` not defined in `Main`
```

```
julia> a = 1;

julia> a
1
```

source

Base.StringIndexError – Type.

```
StringIndexError(str, i)
```

An error occurred when trying to access `str` at index `i` that is not valid.

source

Core.InitError – Type.

```
InitError(mod::Symbol, error)
```

An error occurred when running a module's `__init__` function. The actual error thrown is available in the `.error` field.

source

Base.retry – Function.

```
retry(f;  delays=ExponentialBackOff(), check=nothing) -> Function
```

Return an anonymous function that calls function `f`. If an exception arises, `f` is repeatedly called again, each time `check` returns `true`, after waiting the number of seconds specified in `delays`. `check` should input `delays`'s current state and the `Exception`.

Julia 1.2

Before Julia 1.2 this signature was restricted to `f::Function`.

Examples

```
retry(f, delays=fill(5.0, 3))
retry(f, delays=rand(5:10, 2))
retry(f, delays=Base.ExponentialBackOff(n=3, first_delay=5, max_delay=1000))
retry(http_get, check=(s,e)->e.status == "503")(url)
retry(read, check=(s,e)->isa(e, IOError))(io, 128; all=false)
```

source

Base.ExponentialBackOff – Type.

```
ExponentialBackOff(; n=1, first_delay=0.05, max_delay=10.0, factor=5.0, jitter=0.1)
```

A `Float64` iterator of length `n` whose elements exponentially increase at a rate in the interval `factor * (1 ± jitter)`. The first element is `first_delay` and all elements are clamped to `max_delay`.

source

44.16 Events

Base.Timer – Method.

```
Timer(callback::Function, delay; interval = 0, spawn::Union{Nothing,Bool}=nothing)
```

Create a timer that runs the function `callback` at each timer expiration.

Waiting tasks are woken and the function `callback` is called after an initial delay of `delay` seconds, and then repeating with the given `interval` in seconds. If `interval` is equal to `0`, the callback is only run once. The function `callback` is called with a single argument, the timer itself. Stop a timer by calling `close`. The `callback` may still be run one final time, if the timer has already expired.

If `spawn` is `true`, the created task will be spawned, meaning that it will be allowed to move thread, which avoids the side-effect of forcing the parent task to get stuck to the thread it is on. If `spawn` is `nothing` (default), the task will be spawned if the parent task isn't sticky.

Julia 1.12

The `spawn` argument was introduced in Julia 1.12.

Examples

Here the first number is printed after a delay of two seconds, then the following numbers are printed quickly.

```
julia> begin
           i = 0
           cb(timer) = (global i += 1; println(i))
           t = Timer(cb, 2, interval=0.2)
           wait(t)
           sleep(0.5)
           close(t)
       end
1
2
3
```

source

Base.Timer – Type.

```
Timer(delay; interval = 0)
```

Create a timer that wakes up tasks waiting for it (by calling `wait` on the timer object).

Waiting tasks are woken after an initial delay of at least `delay` seconds, and then repeating after at least `interval` seconds again elapse. If `interval` is equal to `0`, the timer is only triggered once. When the timer is closed (by `close`) waiting tasks are woken with an error. Use `isopen` to check whether a timer is still active. An inactive timer will not fire. Use `t.timeout` and `t.interval` to read the setup conditions of a `Timer` `t`.

```
julia> t = Timer(1.0; interval=0.5)
Timer (open, timeout: 1.0 s, interval: 0.5 s) @0x000000010f4e6e90
```

```
julia> isopen(t)
true

julia> t.timeout
1.0

julia> close(t)

julia> isopen(t)
false
```

Note

`interval` is subject to accumulating time skew. If you need precise events at a particular absolute time, create a new timer at each expiration with the difference to the next time computed.

Note

A `Timer` requires yield points to update its state. For instance, `isopen(t::Timer)` cannot be used to timeout a non-yielding while loop.

!!! compat "Julia 1.12 The `timeout` and `interval` readable properties were added in Julia 1.12.

source

`Base.AsyncCondition` – Type.

```
AsyncCondition()
```

Create an async condition that wakes up tasks waiting for it (by calling `wait` on the object) when notified from C by a call to `uv_async_send`. Waiting tasks are woken with an error when the object is closed (by `close`). Use `isopen` to check whether it is still active. A closed condition is inactive and will not wake up tasks.

This provides an implicit acquire & release memory ordering between the sending and waiting threads.

source

`Base.AsyncCondition` – Method.

```
AsyncCondition(callback::Function)
```

Create an async condition that calls the given `callback` function. The `callback` is passed one argument, the async condition object itself.

source

44.17 Reflection

`Base.nameof` – Method.

```
nameof(m::Module)::Symbol
```

Get the name of a `Module` as a `Symbol`.

Examples

```
julia> nameof(Base.Broadcast)
:Broadcast
```

source

`Base.parentmodule` – Function.

```
parentmodule(m::Method)::Module
```

Return the module in which the given method `m` is defined.

Julia 1.9

Passing a `Method` as an argument requires Julia 1.9 or later.

source

```
parentmodule(f::Function, types)::Module
```

Determine the module containing the first method of a generic function `f` matching the specified `types`.

source

```
parentmodule(f::Function)::Module
```

Determine the module containing the (first) definition of a generic function.

source

```
parentmodule(t::DataType)::Module
```

Determine the module containing the definition of a (potentially `UnionAll`-wrapped) `DataType`.

Examples

```
julia> module Foo
           struct Int end
       end
Foo

julia> parentmodule(Int)
Core

julia> parentmodule(Foo.Int)
Foo
```

source

```
parentmodule(m::Module)::Module
```

Get a module's enclosing `Module`. `Main` is its own parent.

See also: `names`, `nameof`, `fullname`, `@__MODULE__`.

Examples

```
julia> parentmodule(Main)
Main

julia> parentmodule(Base.Broadcast)
Base
```

source

`Base.pathof` – Method.

```
pathof(m::Module)
```

Return the path of the `m.jl` file that was used to `import` module `m`, or `nothing` if `m` was not imported from a package.

Use `dirname` to get the directory part and `basename` to get the file name part of the path.

See also `pkgdir`.

source

`Base.pkgdir` – Method.

```
pkgdir(m::Module[, paths::String...])
```

Return the root directory of the package that declared module `m`, or `nothing` if `m` was not declared in a package. Optionally further path component strings can be provided to construct a path within the package root.

To get the root directory of the package that implements the current module the form `pkgdir(@__MODULE__)` can be used.

If an extension module is given, the root of the parent package is returned.

```
julia> pkgdir(Foo)
"/path/to/Foo.jl"

julia> pkgdir(Foo, "src", "file.jl")
"/path/to/Foo.jl/src/file.jl"
```

See also `pathof`.

Julia 1.7

The optional argument `paths` requires at least Julia 1.7.

source

`Base.pkgversion` – Method.

```
pkgversion(m::Module)
```

If the module `m` belongs to a versioned package, return the version number of that package. Otherwise return `nothing`.

The version is read from the package's Project.toml during package load.

To get the version of the package that imported the current module the form `pkgversion(@__MODULE__)` can be used.

Julia 1.9

This function was introduced in Julia 1.9.

source

`Base.moduleroot` – Function.

```
moduleroot(m::Module)::Module
```

Find the root module of a given module. This is the first module in the chain of parent modules of `m` which is either a registered root module or which is its own parent module.

source

`__module__` – Keyword.

```
__module__
```

The argument `__module__` is only visible inside the macro, and it provides information (in the form of a `Module` object) about the expansion context of the macro invocation. See the manual section on Macro invocation for more information.

source

`__source__` – Keyword.

```
__source__
```

The argument `__source__` is only visible inside the macro, and it provides information (in the form of a `LineNumberNode` object) about the parser location of the `@` sign from the macro invocation. See the manual section on Macro invocation for more information.

source

`Base.@__MODULE__` – Macro.

```
@__MODULE__ -> Module
```

Get the `Module` of the toplevel eval, which is the `Module` code is currently being read from.

source

`Base.@__FUNCTION__` – Macro.

```
@__FUNCTION__
```

Get the innermost enclosing function object.

Note

`@__FUNCTION__` has the same scoping behavior as `return`: when used inside a closure, it refers to the closure and not the outer function. Some macros, including `@spawn`, `@async`, etc., wrap their input in closures. When `@__FUNCTION__` is used within such code, it will refer to the closure created by the macro rather than the enclosing function.

Examples

`@__FUNCTION__` enables recursive anonymous functions:

```
julia> factorial = (n -> n <= 1 ? 1 : n * (@__FUNCTION__)(n - 1));

julia> factorial(5)
120
```

`@__FUNCTION__` can be combined with `nameof` to identify a function's name from within its body:

```
julia> bar() = nameof(@__FUNCTION__);

julia> bar()
:bar
```

Julia 1.13

This macro requires at least Julia 1.13.

source

`Base.@__FILE__` – Macro.

```
@__FILE__ -> String
```

Expand to a string with the path to the file containing the macrocall, or an empty string if evaluated by `julia -e <expr>`. Return `nothing` if the macro was missing parser source information. Alternatively see `PROGRAM_FILE`.

source

`Base.@__DIR__` – Macro.

```
@__DIR__ -> String
```

Macro to obtain the absolute path of the current directory as a string.

If in a script, returns the directory of the script containing the `@__DIR__` macrocall. If run from a REPL or if evaluated by `julia -e <expr>`, returns the current working directory.

Examples

The example illustrates the difference in the behaviors of `@__DIR__` and `pwd()`, by creating a simple script in a different directory than the current working one and executing both commands:

```
julia> cd("/home/JuliaUser") # working directory

julia> # create script at /home/JuliaUser/Projects
       open("/home/JuliaUser/Projects/test.jl","w") do io
           print(io, """
               println("@__DIR__ = ", @__DIR__)
               println("pwd() = ", pwd())
           """)
       end

julia> # outputs script directory and current working directory
       include("/home/JuliaUser/Projects/test.jl")
@__DIR__ = /home/JuliaUser/Projects
pwd() = /home/JuliaUser
```

source

Base.@__LINE__ – Macro.

```
@__LINE__ -> Int
```

Expand to the line number of the location of the macrocall. Return `0` if the line number could not be determined.

source

Base.fullname – Function.

```
fullname(m::Module)
```

Get the fully-qualified name of a module as a tuple of symbols. For example,

Examples

```
julia> fullname(Base.Iterators)
(:Base, :Iterators)

julia> fullname(Main)
(:Main,)
```

source

Base.names – Function.

```
names(x::Module; all::Bool=false, imported::Bool=false, usings::Bool=false,
↪  world::UInt=Base.tls_world_age())::Vector{Symbol}
```

Get a vector of the public names of a `Module`, excluding deprecated names. If `all` is true, then the list also includes non-public names defined in the module, deprecated names, and compiler-generated names. If `imported` is true, then names explicitly imported from other modules are also included. If `usings` is true, then names explicitly or implicitly imported via `using` are also included. Names are returned in sorted order.

As a special case, all names defined in `Main` are considered "public", since it is not idiomatic to explicitly mark names from `Main` as public.

The `world` argument controls the world age used to look up binding partitions, defaulting to the current task's world age. Pass `world=Base.get_world_counter()` to include names from the latest world.

Note

`sym ∈ names(SomeModule)` does *not* imply `isdefined(SomeModule, sym)`. `names` may return symbols marked with `public` or `export`, even if they are not defined in the module.

Warning

`names` may return duplicate names. The duplication happens, e.g. if an `imported` name conflicts with an already existing identifier.

Julia 1.12

The `usings` argument requires Julia 1.12 or later.

See also: `Base.isexported`, `Base.ispublic`, `Base.@locals`, `@__MODULE__`.

source

`Base.isexported` – Function.

```
isexported(m::Module, s::Symbol)::Bool
```

Returns whether a symbol is exported from a module.

See also: `ispublic`, `names`

```
julia> module Mod
           export foo
           public bar
       end
Mod

julia> Base.isexported(Mod, :foo)
true

julia> Base.isexported(Mod, :bar)
false

julia> Base.isexported(Mod, :baz)
false
```

source

`Base.ispublic` – Function.

```
ispublic(m::Module, s::Symbol)::Bool
```

Returns whether a symbol is marked as public in a module.

Exported symbols are considered public.

Julia 1.11

This function and the notion of publicity were added in Julia 1.11.

See also: `isexported`, `names`

```
julia> module Mod
           export foo
           public bar
       end
Mod

julia> Base.ispublic(Mod, :foo)
true

julia> Base.ispublic(Mod, :bar)
true

julia> Base.ispublic(Mod, :baz)
false
```

source

`Base.nameof` – Method.

```
nameof(f::Function)::Symbol
```

Get the name of a generic `Function` as a symbol. For anonymous functions, this is a compiler-generated name. For explicitly-declared subtypes of `Function`, it is the name of the function's type.

source

`Base.functionloc` – Method.

```
functionloc(f::Function, types)
```

Return a tuple `(filename,line)` giving the location of a generic `Function` definition.

source

`Base.functionloc` – Method.

```
functionloc(m::Method)
```

Return a tuple `(filename,line)` giving the location of a `Method` definition.

source

`Base.@locals` – Macro.

```
@locals()
```

Construct a dictionary of the names (as symbols) and values of all local variables defined as of the call site.

Julia 1.1

This macro requires at least Julia 1.1.

Examples

```
julia> let x = 1, y = 2
           Base.@locals
       end
Dict{Symbol, Any} with 2 entries:
  :y => 2
  :x => 1

julia> function f(x)
           local y
           show(Base.@locals); println()
           for i = 1:1
               show(Base.@locals); println()
           end
           y = 2
           show(Base.@locals); println()
           nothing
       end;

julia> f(42)
Dict{Symbol, Any}(:x => 42)
Dict{Symbol, Any}(:i => 1, :x => 42)
Dict{Symbol, Any}(:y => 2, :x => 42)
```

source

`Core.getglobal` – Function.

```
getglobal(module::Module, name::Symbol, [order::Symbol=:monotonic])
```

Retrieve the value of the binding `name` from the module `module`. Optionally, an atomic ordering can be defined for the operation, otherwise it defaults to monotonic.

While accessing module bindings using `getfield` is still supported to maintain compatibility, using `getglobal` should always be preferred since `getglobal` allows for control over atomic ordering (`getfield` is always monotonic) and better signifies the code's intent both to the user as well as the compiler.

Most users should not have to call this function directly – The `getproperty` function or corresponding syntax (i.e. `module.name`) should be preferred in all but few very specific use cases.

Julia 1.9

This function requires Julia 1.9 or later.

See also `getproperty` and `setglobal!`.

Examples

```
julia> a = 1
```

```
1

julia> module M
       a = 2
       end;

julia> getglobal(@__MODULE__, :a)
1

julia> getglobal(M, :a)
2
```

source

`Core.setglobal!` – Function.

```
setglobal!(module::Module, name::Symbol, x, [order::Symbol=:monotonic])
```

Set or change the value of the binding `name` in the module `module` to `x`. No type conversion is performed, so if a type has already been declared for the binding, `x` must be of appropriate type or an error is thrown.

Additionally, an atomic ordering can be specified for this operation, otherwise it defaults to monotonic.

Users will typically access this functionality through the `setproperty!` function or corresponding syntax (i.e. `module.name = x`) instead, so this is intended only for very specific use cases.

Julia 1.9

This function requires Julia 1.9 or later.

See also `setproperty!` and `getglobal`

Examples

```
julia> module M; global a; end;

julia> M.a  # same as `getglobal(M, :a)`
ERROR: UndefVarError: `a` not defined in `M`
Suggestion: add an appropriate import or assignment. This global was declared but not
↪  assigned.
Stacktrace:
 [1] getproperty(x::Module, f::Symbol)
   @ Base ./Base_compiler.jl:40
 [2] top-level scope
   @ none:1

julia> setglobal!(M, :a, 1)
1

julia> M.a
1
```

source

`Core.modifyglobal!` – Function.

```
modifyglobal!(module::Module, name::Symbol, op, x, [order::Symbol=:monotonic])::Pair
```

Atomically perform the operations to get and set a global after applying the function op.

Julia 1.11

This function requires Julia 1.11 or later.

See also `modifyproperty!` and `setglobal!`.

source

`Core.swapglobal!` – Function.

```
swapglobal!(module::Module, name::Symbol, x, [order::Symbol=:monotonic])
```

Atomically perform the operations to simultaneously get and set a global.

Julia 1.11

This function requires Julia 1.11 or later.

See also `swapproperty!` and `setglobal!`.

source

`Core.setglobalonce!` – Function.

```
setglobalonce!(module::Module, name::Symbol, value,
               [success_order::Symbol, [fail_order::Symbol=success_order]) -> success::Bool
```

Atomically perform the operations to set a global to a given value, only if it was previously not set.

Julia 1.11

This function requires Julia 1.11 or later.

See also `setpropertyonce!` and `setglobal!`.

source

`Core.replaceglobal!` – Function.

```
replaceglobal!(module::Module, name::Symbol, expected, desired,
               [success_order::Symbol, [fail_order::Symbol=success_order]) -> (; old,
               ↪  success::Bool)
```

Atomically perform the operations to get and conditionally set a global to a given value.

Julia 1.11

This function requires Julia 1.11 or later.

See also `replaceproperty!` and `setglobal!`.

source

`Core.declare_const` – Function.

```
declare_const(module::Module, name::Symbol, [x])
```

Create or replace the constant `name` in `module` with the new value `x`. When replacing, `x` does not need to have the same type as the original constant.

When `x` is not given, `name` becomes an undefined constant; it cannot be read or written to, but can be redefined.

Unlike the syntax `const`, calling this function does not insert `Core.@latestworld` to update the world age of the current frame:

```
julia> begin
           const x = 1
           println(x)
           const x = 2
           println(x)
           Core.declare_const(Main, :x, 3)
           println(x)
           Core.@latestworld
           println(x)
       end
1
2
2
3
```

Julia 1.12

This function requires Julia 1.12 or later. Redefining constants on earlier versions of Julia is unpredictable.

See also `const`.

source

44.18 Documentation

(See also the documentation chapter.)

`Core.@doc` – Macro.

Documentation

Functions, methods and types can be documented by placing a string before the definition:

```
"""
    foo(x)

Return a fooified version of `x`.
"""
foo(x) = ...
```

The `@doc` macro can be used directly to both set and retrieve documentation / metadata. The macro has special parsing so that the documented object may occur on the next line:

```
@doc "blah"
function foo() ...
```

By default, documentation is written as Markdown, but any object can be used as the first argument.

Documenting objects separately from their definitions

You can document an object before or after its definition with

```
@doc "foo" function_to_doc
@doc "bar" TypeToDoc
```

For macros, the syntax is `@doc "macro doc" :(Module.@macro)` or `@doc "macro doc" :(string_macro"")` for string macros. Without the quote `:()` the expansion of the macro will be documented.

Retrieving Documentation

You can retrieve docs for functions, macros and other objects as follows:

```
@doc foo
@doc @time
@doc md""
```

Julia 1.11

In Julia 1.11 and newer, retrieving documentation with the `@doc` macro requires that the REPL stdlib is loaded.

Functions & Methods

Placing documentation before a method definition (e.g. `function foo() ...` or `foo() = ...`) will cause that specific method to be documented, as opposed to the whole function. Method docs are concatenated together in the order they were defined to provide docs for the function.

source

`Base.Docs.HTML` – Type.

`HTML(s)`: Create an object that renders `s` as html.

```
HTML("<div>foo</div>")
```

You can also use a stream for large amounts of data:

```
HTML() do io
  println(io, "<div>foo</div>")
end
```

Warning

HTML is currently exported to maintain backwards compatibility, but this export is deprecated. It is recommended to use this type as `Docs.HTML` or to explicitly import it from `Docs`.

source

`Base.Docs.Text` – Type.

`Text(s)`: Create an object that renders `s` as plain text.

```
Text("foo")
```

You can also use a stream for large amounts of data:

```
Text() do io
  println(io, "foo")
end
```

Warning

`Text` is currently exported to maintain backwards compatibility, but this export is deprecated. It is recommended to use this type as `Docs.Text` or to explicitly import it from `Docs`.

source

`Base.Docs.hasdoc` – Function.

```
Docs.hasdoc(mod::Module, sym::Symbol)::Bool
```

Return `true` if `sym` in `mod` has a docstring and `false` otherwise.

source

`Base.Docs.undocumented_names` – Function.

```
undocumented_names(mod::Module; private=false)
```

Return a sorted vector of undocumented symbols in `module` (that is, lacking docstrings). `private=false` (the default) returns only identifiers declared with `public` and/or `export`, whereas `private=true` returns all symbols in the module (excluding compiler-generated hidden symbols starting with #).

See also: `names`, `Docs.hasdoc`, `Base.ispublic`.

source

44.19 Code loading

`Base.identify_package` – Function.

```
Base.identify_package(name::String)::Union{PkgId, Nothing}
Base.identify_package(where::Union{Module,PkgId}, name::String)::Union{PkgId, Nothing}
```

Identify the package by its name from the current environment stack, returning its `PkgId`, or `nothing` if it cannot be found.

If only the `name` argument is provided, it searches each environment in the stack and its named direct dependencies.

The `where` argument provides the context from where to search for the package: in this case it first checks if the name matches the context itself, otherwise it searches all recursive dependencies (from the resolved manifest of each environment) until it locates the context `where`, and from there identifies the dependency with the corresponding name.

```
julia> Base.identify_package("Pkg") # Pkg is a dependency of the default environment
Pkg [44cfe95a-1eb2-52ea-b672-e2afdf69b78f]

julia> using LinearAlgebra

julia> Base.identify_package(LinearAlgebra, "Pkg") # Pkg is not a dependency of LinearAlgebra
```

source

Base.locate_package – Function.

```
Base.locate_package(pkg::PkgId)::Union{String, Nothing}
```

The path to the entry-point file for the package corresponding to the identifier pkg, or nothing if not found. See also identify_package.

```
julia> pkg = Base.identify_package("Pkg")
Pkg [44cfe95a-1eb2-52ea-b672-e2afdf69b78f]

julia> Base.locate_package(pkg)
"/path/to/julia/stdlib/v1.13/Pkg/src/Pkg.jl"
```

source

Base.require – Function.

```
require(into::Module, module::Symbol)
```

This function is part of the implementation of using / import, if a module is not already defined in Main. It can also be called directly to force reloading a module, regardless of whether it has been loaded before (for example, when interactively developing libraries).

Loads a source file, in the context of the Main module, on every active node, searching standard locations for files. require is considered a top-level operation, so it sets the current include path but does not use it to search for files (see help for include). This function is typically used to load library code, and is implicitly called by using to load packages.

When searching for files, require first looks for package code in the global array LOAD_PATH. require is case-sensitive on all platforms, including those with case-insensitive filesystems like macOS and Windows.

For more details regarding code loading, see the manual sections on modules and parallel computing.

source

Base.compilecache – Function.

```
Base.compilecache(module::PkgId)
```

Creates a precompiled cache file for a module and all of its dependencies. This can be used to reduce package load times. Cache files are stored in DEPOT_PATH[1]/compiled. See Module initialization and precompilation for important notes.

source

`Base.isprecompiled` – Function.

```
Base.isprecompiled(pkg::PkgId; ignore_loaded::Bool=false)
```

Returns whether a given PkgId within the active project is precompiled.

By default this check observes the same approach that code loading takes with respect to when different versions of dependencies are currently loaded to that which is expected. To ignore loaded modules and answer as if in a fresh julia session specify `ignore_loaded=true`.

Julia 1.10

This function requires at least Julia 1.10.

source

`Base.get_extension` – Function.

```
get_extension(parent::Module, extension::Symbol)
```

Return the module for `extension` of `parent` or return `nothing` if the extension is not loaded.

source

44.20 Internals

`Base.GC.gc` – Function.

```
GC.gc([full=true])
```

Perform garbage collection. The argument `full` determines the kind of collection: a full collection (default) traverses all live objects (i.e. full mark) and should reclaim memory from all unreachable objects. An incremental collection only reclaims memory from young objects which are not reachable.

The GC may decide to perform a full collection even if an incremental collection was requested.

Warning

Excessive use will likely lead to poor performance.

source

`Base.GC.enable` – Function.

```
GC.enable(on::Bool)
```

Control whether garbage collection is enabled using a boolean argument (`true` for enabled, `false` for disabled). Return previous GC state.

Warning

Disabling garbage collection should be used only with caution, as it can cause memory use to grow without bound.

source

`Base.GC.@preserve` – Macro.

```
GC.@preserve x1 x2 ... xn expr
```

Mark the objects `x1, x2, ...` as being *in use* during the evaluation of the expression `expr`. This is only required in unsafe code where `expr` *implicitly uses* memory or other resources owned by one of the xs.

Implicit use of `x` covers any indirect use of resources logically owned by `x` which the compiler cannot see. Some examples:

- Accessing memory of an object directly via a `Ptr`
- Passing a pointer to `x` to `ccall`
- Using resources of `x` which would be cleaned up in the finalizer.

`@preserve` should generally not have any performance impact in typical use cases where it briefly extends object lifetime. In implementation, `@preserve` has effects such as protecting dynamically allocated objects from garbage collection.

Examples

When loading from a pointer with `unsafe_load`, the underlying object is implicitly used, for example `x` is implicitly used by `unsafe_load(p)` in the following:

```
julia> let
           x = Ref{Int}(101)
           p = Base.unsafe_convert(Ptr{Int}, x)
           GC.@preserve x unsafe_load(p)
       end
101
```

When passing pointers to `ccall`, the pointed-to object is implicitly used and should be preserved. (Note however that you should normally just pass `x` directly to `ccall` which counts as an explicit use.)

```
julia> let
           x = "Hello"
           p = pointer(x)
           Int(GC.@preserve x @ccall strlen(p::Cstring)::Csize_t)
           # Preferred alternative
           Int(@ccall strlen(x::Cstring)::Csize_t)
       end
5
```

source

`Base.GC.safepoint` – Function.

```
GC.safepoint()
```

Inserts a point in the program where garbage collection may run.

Safepoints are fast and do not themselves trigger garbage collection. However, if another thread has requested the GC to run, reaching a safepoint will cause the current thread to block and wait for the GC.

This can be useful in rare cases in multi-threaded programs where some tasks are allocating memory (and hence may need to run GC) but other tasks are doing only simple operations (no allocation, task switches, or I/O), which do not yield control to Julia's runtime, and therefore blocks the GC from running. Calling this function periodically in the non-allocating tasks allows garbage collection to run.

Note that even though safepoints are fast (typically around 2 clock cycles), they can still degrade performance if called in a tight loop.

Julia 1.4

This function is available as of Julia 1.4.

source

`Base.GC.enable_logging` – Function.

```
GC.enable_logging(on::Bool)
```

When turned on, print statistics about each GC to stderr.

source

`Base.GC.logging_enabled` – Function.

```
GC.logging_enabled()
```

Return whether GC logging has been enabled via `GC.enable_logging`.

source

`Base.Meta.lower` – Function.

```
lower(m, x)
```

Takes the expression `x` and returns an equivalent expression in lowered form for executing in module `m`. See also `code_lowered`.

source

`Base.Meta.@lower` – Macro.

```
@lower [m] x
```

Return lowered form of the expression `x` in module `m`. By default `m` is the module in which the macro is called. See also `lower`.

source

`Base.Meta.parse` – Method.

```
parse(str, start; greedy=true, raise=true, depwarn=true, filename="none")
```

Parse the expression string and return an expression (which could later be passed to eval for execution). `start` is the code unit index into `str` of the first character to start parsing at (as with all string indexing, these are not character indices). If `greedy` is `true` (default), `parse` will try to consume as much input as it can; otherwise, it will stop as soon as it has parsed a valid expression. Incomplete but otherwise

syntactically valid expressions will return `Expr(:incomplete, "(error message)")`. If `raise` is `true` (default), syntax errors other than incomplete expressions will raise an error. If `raise` is `false`, `parse` will return an expression that will raise an error upon evaluation. If `depwarn` is `false`, deprecation warnings will be suppressed. The `filename` argument is used to display diagnostics when an error is raised.

```
julia> Meta.parse("(α, β) = 3, 5", 1) # start of string
(:((α, β) = (3, 5)), 16)

julia> Meta.parse("(α, β) = 3, 5", 1, greedy=false)
(:((α, β)), 9)

julia> Meta.parse("(α, β) = 3, 5", 16) # end of string
(nothing, 16)

julia> Meta.parse("(α, β) = 3, 5", 11) # index of 3
(:((3, 5)), 16)

julia> Meta.parse("(α, β) = 3, 5", 11, greedy=false)
(3, 13)
```

source

`Base.Meta.parse` – Method.

```
parse(str; raise=true, depwarn=true, filename="none")
```

Parse the expression string greedily, returning a single expression. An error is thrown if there are additional characters after the first expression. If `raise` is `true` (default), syntax errors will raise an error; otherwise, `parse` will return an expression that will raise an error upon evaluation. If `depwarn` is `false`, deprecation warnings will be suppressed. The `filename` argument is used to display diagnostics when an error is raised.

```
julia> Meta.parse("x = 3")
:(x = 3)

julia> Meta.parse("1.0.2")
ERROR: ParseError:
# Error @ none:1:1
1.0.2
└──┘ ── invalid numeric constant
[...]

julia> Meta.parse("1.0.2"; raise = false)
:($(Expr(:error, "invalid numeric constant "1.0."")))

julia> Meta.parse("x = ")
:($(Expr(:incomplete, "incomplete: premature end of input")))
```

source

`Base.Meta.ParseError` – Type.

```
ParseError(msg)
```

The expression passed to the `parse` function could not be interpreted as a valid Julia expression.

source

`Core.QuoteNode` – Type.

```
QuoteNode
```

A quoted piece of code, that does not support interpolation. See the manual section about QuoteNodes for details.

source

`Base.macroexpand` – Function.

```
macroexpand(m::Module, x; recursive=true, legacyscope=true)
```

Take the expression `x` and return an equivalent expression with all macros removed (expanded) for executing in module `m`. The `recursive` keyword controls whether deeper levels of nested macros are also expanded. The `legacyscope` keyword controls whether legacy macroscope expansion is performed. This is demonstrated in the example below:

```
julia> module M
           macro m1()
               42
           end
           macro m2()
               :(@m1())
           end
       end
M

julia> macroexpand(M, :(@m2()), recursive=true)
42

julia> macroexpand(M, :(@m2()), recursive=false)
:(#= REPL[1]:6 =# @m1)
```

Julia 1.13

The `legacyscope` keyword argument requires at least Julia 1.13.

source

`Base.macroexpand!` – Function.

```
macroexpand!(m::Module, x; recursive=true, legacyscope=false)
```

Take the expression `x` and return an equivalent expression with all macros removed (expanded) for executing in module `m`, modifying `x` in place without copying. The `recursive` keyword controls whether deeper levels of nested macros are also expanded. The `legacyscope` keyword controls whether legacy macroscope expansion is performed.

This function performs macro expansion without the initial copy step, making it more efficient when the original expression is no longer needed. By default, macroscope expansion is disabled for in-place expansion as it can be called separately if needed.

Warning

This function modifies the input expression x in place. Use `macroexpand` if you need to preserve the original expression.

Julia 1.13

This function requires at least Julia 1.13.

source

`Base.@macroexpand` – Macro.

```
@macroexpand [mod,] ex
```

Return equivalent expression with all macros removed (expanded). If two arguments are provided, the first is the module to evaluate in.

There are differences between `@macroexpand` and `macroexpand`.

- While `macroexpand` takes a keyword argument `recursive`, `@macroexpand` is always recursive. For a non recursive macro version, see `@macroexpand1`.
- While `macroexpand` has an explicit `module` argument, `@macroexpand` always expands with respect to the module in which it is called.

This is best seen in the following example:

```
julia> module M
           macro m()
               1
           end
           function f()
               (@macroexpand(@m),
                macroexpand(M, :(@m)),
                macroexpand(parentmodule(M), :(@m))
               )
           end
       end
M

julia> macro m()
           2
       end
@m (macro with 1 method)

julia> M.f()
(1, 1, 2)
```

With `@macroexpand` the expression expands where `@macroexpand` appears in the code (module `M` in the example). With `macroexpand` the expression expands in the module given as the first argument.

Julia 1.11

The two-argument form requires at least Julia 1.11.

source

`Base.@macroexpand1` – Macro.

```
@macroexpand1 [mod,] ex
```

Non recursive version of `@macroexpand`.

source

`Base.code_lowered` – Function.

```
code_lowered(f, types; generated=true, debuginfo=:default)
```

Return an array of the lowered forms (IR) for the methods matching the given generic function and type signature.

If `generated` is `false`, the returned `CodeInfo` instances will correspond to fallback implementations. An error is thrown if no fallback implementation exists. If `generated` is `true`, these `CodeInfo` instances will correspond to the method bodies yielded by expanding the generators.

The keyword `debuginfo` controls the amount of code metadata present in the output.

Note that an error will be thrown if `types` are not concrete types when `generated` is `true` and any of the corresponding methods are an `@generated` method.

source

`Base.code_typed` – Function.

```
code_typed(f, types; kw...)
```

Returns an array of type-inferred lowered form (IR) for the methods matching the given generic function and type signature.

Keyword Arguments

- `optimize::Bool = true`: optional, controls whether additional optimizations, such as inlining, are also applied.
- `debuginfo::Symbol = :default`: optional, controls the amount of code metadata present in the output, possible options are `:source` or `:none`.

Internal Keyword Arguments

This section should be considered internal, and is only for who understands Julia compiler internals.

- `world::UInt = Base.get_world_counter()`: optional, controls the world age to use when looking up methods, use current world age if not specified.

- `interp::Core.Compiler.AbstractInterpreter = Core.Compiler.NativeInterpreter(world)`: optional, controls the abstract interpreter to use, use the native interpreter if not specified.

Examples

One can put the argument types in a tuple to get the corresponding `code_typed`.

```
julia> code_typed(+, (Float64, Float64))
1-element Vector{Any}:
 CodeInfo(
1 — %1 = Base.add_float(x, y)::Float64
└──      return %1
) => Float64

julia> code_typed((typeof(-), Float64, Float64))
1-element Vector{Any}:
 CodeInfo(
1 — %1 = Base.sub_float(x, y)::Float64
└──      return %1
) => Float64

julia> code_typed((Type{Int}, UInt8))
1-element Vector{Any}:
 CodeInfo(
1 — %1 = Core.zext_int(Core.Int64, x)::Int64
└──      return %1
) => Int64

julia> code_typed((Returns{Int64},))
1-element Vector{Any}:
 CodeInfo(
1 — %1 =   builtin Base.getfield(obj, :value)::Int64
└──      return %1
) => Int64
```

source

`Base.precompile` – Function.

```
precompile(f, argtypes::Tuple{Vararg{Any}}, m::Method)
```

Precompile a specific method for the given argument types. This may be used to precompile a different method than the one that would ordinarily be chosen by dispatch, thus mimicking `invoke`.

source

```
precompile(f, argtypes::Tuple{Vararg{Any}})
```

Compile the given function `f` for the argument tuple (of types) `argtypes`, but do not execute it.

source

`Base.jit_total_bytes` – Function.

```
Base.jit_total_bytes()
```

Return the total amount (in bytes) allocated by the just-in-time compiler for e.g. native code and data.

source

44.21 Meta

Base.Meta.quot – Function.

```
Meta.quot(ex)::Expr
```

Quote expression ex to produce an expression with head quote. This can for instance be used to represent objects of type Expr in the AST. See also the manual section about QuoteNode.

Examples

```
julia> eval(Meta.quot(:x))
:x

julia> dump(Meta.quot(:x))
Expr
  head: Symbol quote
  args: Array{Any}((1,))
    1: Symbol x

julia> eval(Meta.quot(:(1+2)))
:(1 + 2)
```

source

Base.isexpr – Function.

```
Meta.isexpr(ex, head[, n])::Bool
```

Return true if ex is an Expr with the given type head and optionally that the argument list is of length n. head may be a Symbol or collection of Symbols. For example, to check that a macro was passed a function call expression, you might use isexpr(ex, :call).

Examples

```
julia> ex = :(f(x))
:(f(x))

julia> Meta.isexpr(ex, :block)
false

julia> Meta.isexpr(ex, :call)
true

julia> Meta.isexpr(ex, [:block, :call]) # multiple possible heads
true

julia> Meta.isexpr(ex, :call, 1)
false
```

```
julia> Meta.isexpr(ex, :call, 2)
true
```

source

Base.Meta.isidentifier – Function.

```
isidentifier(s) -> Bool
```

Return whether the symbol or string s contains characters that are parsed as a valid ordinary identifier (not a binary/unary operator) in Julia code; see also Base.isoperator.

Internally Julia allows any sequence of characters in a Symbol (except \0s), and macros automatically use variable names containing # in order to avoid naming collision with the surrounding code. In order for the parser to recognize a variable, it uses a limited set of characters (greatly extended by Unicode). isidentifier() makes it possible to query the parser directly whether a symbol contains valid characters.

Examples

```
julia> Meta.isidentifier(:x), Meta.isidentifier("1x")
(true, false)
```

source

Base.Meta.isoperator – Function.

```
isoperator(s::Symbol)
```

Return true if the symbol can be used as an operator, false otherwise.

Examples

```
julia> Meta.isoperator(:+), Meta.isoperator(:f)
(true, false)
```

source

Base.Meta.isunaryoperator – Function.

```
isunaryoperator(s::Symbol)
```

Return true if the symbol can be used as a unary (prefix) operator, false otherwise.

Examples

```
julia> Meta.isunaryoperator(:-), Meta.isunaryoperator(:√), Meta.isunaryoperator(:f)
(true, true, false)
```

source

Base.Meta.isbinaryoperator – Function.

```
isbinaryoperator(s::Symbol)
```

Return `true` if the symbol can be used as a binary (infix) operator, `false` otherwise.

Examples

```
julia> Meta.isbinaryoperator(:-), Meta.isbinaryoperator(:√), Meta.isbinaryoperator(:f)
(true, false, false)
```

source

`Base.Meta.show_sexpr` – Function.

```
Meta.show_sexpr([io::IO,], ex)
```

Show expression ex as a lisp style S-expression.

Examples

```
julia> Meta.show_sexpr(:(f(x, g(y,z))))
(:call, :f, :x, (:call, :g, :y, :z))
```

source

Chapter 45

Collections and Data Structures

45.1 Iteration

Sequential iteration is implemented by the `iterate` function. The general `for` loop:

```
for i in iter   # or  "for i = iter"
    # body
end
```

is translated into:

```
next = iterate(iter)
while next !== nothing
    (i, state) = next
    # body
    next = iterate(iter, state)
end
```

The `state` object may be anything, and should be chosen appropriately for each iterable type. See the manual section on the iteration interface for more details about defining a custom iterable type.

`Base.iterate` – Function.

```
iterate(iter [, state])::Union{Nothing, Tuple{Any, Any}}
```

Advance the iterator to obtain the next element. If no elements remain, `nothing` should be returned. Otherwise, a 2-tuple of the next element and the new iteration state should be returned.

source

`Base.IteratorSize` – Type.

```
IteratorSize(itertype::Type)::IteratorSize
```

Given the type of an iterator, return one of the following values:

- `SizeUnknown()` if the length (number of elements) cannot be determined in advance.

- `HasLength()` if there is a fixed, finite length.
- `HasShape{N}()` if there is a known length plus a notion of multidimensional shape (as for an array). In this case `N` should give the number of dimensions, and the `axes` function is valid for the iterator.
- `IsInfinite()` if the iterator yields values forever.

The default value (for iterators that do not define this function) is `HasLength()`. This means that most iterators are assumed to implement `length`.

This trait is generally used to select between algorithms that pre-allocate space for their result, and algorithms that resize their result incrementally.

```
julia> Base.IteratorSize(1:5)
Base.HasShape{1}()

julia> Base.IteratorSize((2,3))
Base.HasLength()
```

source

`Base.IteratorEltype` – Type.

```
IteratorEltype(itertype::Type)::IteratorEltype
```

Given the type of an iterator, return one of the following values:

- `EltypeUnknown()` if the type of elements yielded by the iterator is not known in advance.
- `HasEltype()` if the element type is known, and `eltype` would return a meaningful value.

`HasEltype()` is the default, since iterators are assumed to implement `eltype`.

This trait is generally used to select between algorithms that pre-allocate a specific type of result, and algorithms that pick a result type based on the types of yielded values.

```
julia> Base.IteratorEltype(1:5)
Base.HasEltype()
```

source

Fully implemented by:

- `AbstractRange`
- `UnitRange`
- `Tuple`
- `Number`
- `AbstractArray`
- `BitSet`

- IdDict
- Dict
- WeakKeyDict
- EachLine
- AbstractString
- Set
- Pair
- NamedTuple

45.2 Constructors and Types

Base.AbstractRange – Type.

```
AbstractRange{T} <: AbstractVector{T}
```

Supertype for linear ranges with elements of type T. UnitRange, LinRange and other types are subtypes of this.

All subtypes must define step. Thus LogRange is not a subtype of AbstractRange.

source

Base.OrdinalRange – Type.

```
OrdinalRange{T, S} <: AbstractRange{T}
```

Supertype for ordinal ranges with elements of type T with spacing(s) of type S. The steps should be always-exact multiples of oneunit, and T should be a "discrete" type, which cannot have values smaller than oneunit. For example, Integer or Date types would qualify, whereas Float64 would not (since this type can represent values smaller than oneunit(Float64). UnitRange, StepRange, and other types are subtypes of this.

source

Base.AbstractUnitRange – Type.

```
AbstractUnitRange{T} <: OrdinalRange{T, T}
```

Supertype for ranges with a step size of oneunit(T) with elements of type T. UnitRange and other types are subtypes of this.

source

Base.StepRange – Type.

```
StepRange{T, S} <: OrdinalRange{T, S}
```

Ranges with elements of type T with spacing of type S. The step between each element is constant, and the range is defined in terms of a `start` and `stop` of type T and a `step` of type S. Neither T nor S should be floating point types. The syntax `a:b:c` with `b != 0` and a, b, and c all integers creates a StepRange.

Examples

```
julia> collect(StepRange(1, Int8(2), 10))
5-element Vector{Int64}:
 1
 3
 5
 7
 9

julia> typeof(StepRange(1, Int8(2), 10))
StepRange{Int64, Int8}

julia> typeof(1:3:6)
StepRange{Int64, Int64}
```

source

`Base.UnitRange` – Type.

```
UnitRange{T<:Real}
```

A range parameterized by a `start` and `stop` of type T, filled with elements spaced by 1 from `start` until `stop` is exceeded. The syntax `a:b` with a and b both `Integers` creates a `UnitRange`.

Examples

```
julia> collect(UnitRange(2.3, 5.2))
3-element Vector{Float64}:
 2.3
 3.3
 4.3

julia> typeof(1:10)
UnitRange{Int64}
```

source

`Base.AbstractOneTo` – Type.

```
Base.AbstractOneTo
```

Abstract type for ranges that start at 1 and have a step size of 1.

Julia 1.13

This type requires at least Julia 1.13.

source

Base.LinRange – Type.

```
LinRange{T,L}
```

A range with `len` linearly spaced elements between its `start` and `stop`. The size of the spacing is controlled by `len`, which must be an `Integer`.

Examples

```
julia> LinRange(1.5, 5.5, 9)
9-element LinRange{Float64, Int64}:
 1.5, 2.0, 2.5, 3.0, 3.5, 4.0, 4.5, 5.0, 5.5
```

Compared to using `range`, directly constructing a `LinRange` should have less overhead but won't try to correct for floating point errors:

```
julia> collect(range(-0.1, 0.3, length=5))
5-element Vector{Float64}:
 -0.1
  0.0
  0.1
  0.2
  0.3

julia> collect(LinRange(-0.1, 0.3, 5))
5-element Vector{Float64}:
 -0.1
 -1.3877787807814457e-17
  0.09999999999999999
  0.19999999999999998
  0.3
```

See also `Base.LogRange` for logarithmically spaced points.

source

45.3 General Collections

Base.isempty – Function.

```
isempty(condition)
```

Return `true` if no tasks are waiting on the condition, `false` otherwise.

source

```
isempty(collection)::Bool
```

Determine whether a collection is empty (has no elements).

Warning

`isempty(itr)` may consume the next element of a stateful iterator `itr` unless an appropriate `Base.isdone(itr)` method is defined. Stateful iterators *should* implement `isdone`, but you may want to avoid using `isempty` when writing generic code which should support any iterator type.

Examples

```
julia> isempty([])
true

julia> isempty([1 2 3])
false
```

source

Base.isdone – Function.

```
isdone(itr, [state])::Union{Bool, Missing}
```

This function provides a fast-path hint for iterator completion. This is useful for stateful iterators that want to avoid having elements consumed if they are not going to be exposed to the user (e.g. when checking for done-ness in `isempty` or `zip`).

Stateful iterators that want to opt into this feature should define an `isdone` method that returns true/false depending on whether the iterator is done or not. Stateless iterators need not implement this function.

If the result is `missing`, then `isdone` cannot determine whether the iterator state is terminal, and callers must compute `iterate(itr, state) === nothing` to obtain a definitive answer.

See also `iterate`, `isempty`

source

Base.empty! – Function.

```
empty!(c::Channel)
```

Empty a Channel `c` by calling `empty!` on the internal buffer. Return the empty channel.

source

```
empty!(collection) -> collection
```

Remove all elements from a `collection`.

Examples

```
julia> A = Dict("a" => 1, "b" => 2)
Dict{String, Int64} with 2 entries:
  "b" => 2
  "a" => 1
```

```
julia> empty!(A);

julia> A
Dict{String, Int64}()
```

source

Base.length – Function.

```
length(collection)::Integer
```

Return the number of elements in the collection.

Use lastindex to get the last valid index of an indexable collection.

See also: size, ndims, eachindex.

Examples

```
julia> length(1:5)
5

julia> length([1, 2, 3, 4])
4

julia> length([1 2; 3 4])
4
```

source

Base.checked_length – Function.

```
Base.checked_length(r)
```

Calculates length(r), but may check for overflow errors where applicable when the result doesn't fit into Union{Integer(eltype(r)),Int}.

source

Fully implemented by:

- AbstractRange
- UnitRange
- Tuple
- Number
- AbstractArray
- BitSet
- IdDict
- Dict

- WeakKeyDict
- AbstractString
- Set
- NamedTuple

45.4 Iterable Collections

Base.in – Function.

```
in(item, collection)::Bool
∈(item, collection)::Bool
```

Determine whether an item is in the given collection, in the sense that it is == to one of the values generated by iterating over the collection. Can equivalently be used with infix syntax:

```
item in collection
item ∈ collection
```

Return a Bool value, except if item is missing or collection contains missing but not item, in which case missing is returned (three-valued logic, matching the behavior of any and ==). Some collections follow a slightly different definition. For example, Sets check whether the item isequal to one of the elements; Dicts look for key=>value pairs, and the key is compared using isequal.

To test for the presence of a key in a dictionary, use haskey or k in keys(dict). For the collections mentioned above, the result is always a Bool.

When broadcasting with in.(items, collection) or items .∈ collection, both items and collection are broadcasted over, which is often not what is intended. For example, if both arguments are vectors (and the dimensions match), the result is a vector indicating whether each value in collection items is in the value at the corresponding position in collection. To get a vector indicating whether each value in items is in collection, wrap collection in a tuple or a Ref like this: in.(items, Ref(collection)) or items .∈ Ref(collection).

See also: ∉, insorted, contains, occursin, issubset.

Examples

```
julia> a = 1:3:20
1:3:19

julia> 4 in a
true

julia> 5 in a
false

julia> missing in [1, 2]
missing

julia> 1 in [2, missing]
missing
```

```
julia> 1 in [1, missing]
true

julia> missing in Set([1, 2])
false

julia> (1=>missing) in Dict(1=>10, 2=>20)
missing

julia> [1, 2] .∈ [2, 3]
2-element BitVector:
 0
 0

julia> [1, 2] .∈ ([2, 3],)
2-element BitVector:
 0
 1
```

source

Base.:∉ – Function.

```
∉(item, collection)::Bool
∌(collection, item)::Bool
```

Negation of ∈ and ∋, i.e. checks that `item` is not in `collection`.

When broadcasting with `items .∉ collection`, both `items` and `collection` are broadcasted over, which is often not what is intended. For example, if both arguments are vectors (and the dimensions match), the result is a vector indicating whether each value in collection `items` is not in the value at the corresponding position in `collection`. To get a vector indicating whether each value in `items` is not in `collection`, wrap `collection` in a tuple or a `Ref` like this: `items .∉ Ref(collection)`.

Examples

```
julia> 1 ∉ 2:4
true

julia> 1 ∉ 1:3
false

julia> [1, 2] .∉ [2, 3]
2-element BitVector:
 1
 1

julia> [1, 2] .∉ ([2, 3],)
2-element BitVector:
 1
 0
```

source

Base.hasfastin – Function.

```
Base.hasfastin(T)
```

Determine whether the computation `x ∈ collection` where `collection::T` can be considered as a "fast" operation (typically constant or logarithmic complexity). The definition `hasfastin(x) = hasfastin(typeof(x))` is provided for convenience so that instances can be passed instead of types. However the form that accepts a type argument should be defined for new types.

The default for `hasfastin(T)` is `true` for subtypes of `AbstractSet`, `AbstractDict` and `AbstractRange` and `false` otherwise.

source

Base.eltype – Function.

```
eltype(type)
```

Determine the type of the elements generated by iterating a collection of the given `type`. For dictionary types, this will be a `Pair{KeyType,ValType}`. The definition `eltype(x) = eltype(typeof(x))` is provided for convenience so that instances can be passed instead of types. However the form that accepts a type argument should be defined for new types.

See also: `keytype`, `typeof`.

Examples

```
julia> eltype(fill(1f0, (2,2)))
Float32

julia> eltype(fill(0x1, (2,2)))
UInt8
```

source

Base.indexin – Function.

```
indexin(a, b)
```

Return an array containing the first index in `b` for each value in `a` that is a member of `b`. The output array contains `nothing` wherever `a` is not a member of `b`.

See also: `sortperm`, `findfirst`.

Examples

```
julia> a = ['a', 'b', 'c', 'b', 'd', 'a'];

julia> b = ['a', 'b', 'c'];

julia> indexin(a, b)
6-element Vector{Union{Nothing, Int64}}:
 1
 2
 3
```

```
 2
  nothing
 1

julia> indexin(b, a)
3-element Vector{Union{Nothing, Int64}}:
 1
 2
 3
```

source

Base.unique – Function.

```
unique(A::AbstractArray; dims::Int)
```

Return unique regions of A along dimension dims.

Examples

```
julia> A = map(isodd, reshape(Vector(1:8), (2,2,2)))
2×2×2 Array{Bool, 3}:
[:, :, 1] =
 1  1
 0  0

[:, :, 2] =
 1  1
 0  0

julia> unique(A)
2-element Vector{Bool}:
 1
 0

julia> unique(A, dims=2)
2×1×2 Array{Bool, 3}:
[:, :, 1] =
 1
 0

[:, :, 2] =
 1
 0

julia> unique(A, dims=3)
2×2×1 Array{Bool, 3}:
[:, :, 1] =
 1  1
 0  0
```

source

```
unique(f, itr)
```

Return an array containing one value from `itr` for each unique value produced by `f` applied to elements of `itr`.

Examples

```
julia> unique(x -> x^2, [1, -1, 3, -3, 4])
3-element Vector{Int64}:
 1
 3
 4
```

This functionality can also be used to extract the *indices* of the first occurrences of unique elements in an array:

```
julia> a = [3.1, 4.2, 5.3, 3.1, 3.1, 3.1, 4.2, 1.7];

julia> i = unique(i -> a[i], eachindex(a))
4-element Vector{Int64}:
 1
 2
 3
 8

julia> a[i]
4-element Vector{Float64}:
 3.1
 4.2
 5.3
 1.7

julia> a[i] == unique(a)
true
```

source

```
unique(itr)
```

Return an array containing only the unique elements of collection `itr`, as determined by `isequal` and `hash`, in the order that the first of each set of equivalent elements originally appears. The element type of the input is preserved.

See also: `unique!`, `allunique`, `allequal`.

Examples

```
julia> unique([1, 2, 6, 2])
3-element Vector{Int64}:
 1
 2
 6

julia> unique(Real[1, 1.0, 2])
2-element Vector{Real}:
 1
 2
```

source

Base.unique! – Function.

```
unique!(A::AbstractVector)
```

Remove duplicate items as determined by isequal and hash, then return the modified A. unique! will return the elements of A in the order that they occur. If you do not care about the order of the returned data, then calling (sort!(A); unique!(A)) will be much more efficient as long as the elements of A can be sorted.

Examples

```
julia> unique!([1, 1, 1])
1-element Vector{Int64}:
 1

julia> A = [7, 3, 2, 3, 7, 5];

julia> unique!(A)
4-element Vector{Int64}:
 7
 3
 2
 5

julia> B = [7, 6, 42, 6, 7, 42];

julia> sort!(B);  # unique! is able to process sorted data much more efficiently.

julia> unique!(B)
3-element Vector{Int64}:
  6
  7
 42
```

source

```
unique!(f, A::AbstractVector)
```

Selects one value from A for each unique value produced by f applied to elements of A, then return the modified A.

Julia 1.1

This method is available as of Julia 1.1.

Examples

```
julia> unique!(x -> x^2, [1, -1, 3, -3, 4])
3-element Vector{Int64}:
 1
 3
```

```
 4

julia> unique!(n -> n%3, [5, 1, 8, 9, 3, 4, 10, 7, 2, 6])
3-element Vector{Int64}:
 5
 1
 9

julia> unique!(iseven, [2, 3, 5, 7, 9])
2-element Vector{Int64}:
 2
 3
```

source

Base.allunique – Function.

```
allunique(itr)::Bool
allunique(f, itr)::Bool
```

Return `true` if all values from `itr` are distinct when compared with `isequal`. Or if all of `[f(x) for x in itr]` are distinct, for the second method.

Note that `allunique(f, itr)` may call `f` fewer than `length(itr)` times. The precise number of calls is regarded as an implementation detail.

`allunique` may use a specialized implementation when the input is sorted.

See also: `unique`, `issorted`, `allequal`.

Julia 1.11

The method `allunique(f, itr)` requires at least Julia 1.11.

Examples

```
julia> allunique([1, 2, 3])
true

julia> allunique([1, 2, 1, 2])
false

julia> allunique(Real[1, 1.0, 2])
false

julia> allunique([NaN, 2.0, NaN, 4.0])
false

julia> allunique(abs, [1, -1, 2])
false
```

source

Base.allequal – Function.

```
allequal(itr)::Bool
allequal(f, itr)::Bool
```

Return `true` if all values from `itr` are equal when compared with `isequal`. Or if all of `[f(x) for x in itr]` are equal, for the second method.

Note that `allequal(f, itr)` may call `f` fewer than `length(itr)` times. The precise number of calls is regarded as an implementation detail.

See also: `unique`, `allunique`.

Julia 1.8

The `allequal` function requires at least Julia 1.8.

Julia 1.11

The method `allequal(f, itr)` requires at least Julia 1.11.

Examples

```
julia> allequal([])
true

julia> allequal([1])
true

julia> allequal([1, 1])
true

julia> allequal([1, 2])
false

julia> allequal(Dict(:a => 1, :b => 1))
false

julia> allequal(abs2, [1, -1])
true
```

source

`Base.reduce` – Method.

```
reduce(op, itr; [init])
```

Reduce the given collection `itr` with the given binary operator `op`. If provided, the initial value `init` must be a neutral element for `op` that will be returned for empty collections. It is unspecified whether `init` is used for non-empty collections.

For empty collections, providing `init` will be necessary, except for some special cases (e.g. when `op` is one of `+`, `*`, `max`, `min`, `&`, `|`) when Julia can determine the neutral element of `op`.

Reductions for certain commonly-used operators may have special implementations, and should be used instead: `maximum(itr)`, `minimum(itr)`, `sum(itr)`, `prod(itr)`, `any(itr)`, `all(itr)`. There are efficient methods for concatenating certain arrays of arrays by calling `reduce(vcat, arr)` or `reduce(hcat, arr)`.

The associativity of the reduction is implementation dependent. This means that you can't use non-associative operations like `-` because it is undefined whether `reduce(-,[1,2,3])` should be evaluated as `(1-2)-3` or `1-(2-3)`. Use `foldl` or `foldr` instead for guaranteed left or right associativity.

Some operations accumulate error. Parallelism will be easier if the reduction can be executed in groups. Future versions of Julia might change the algorithm. Note that the elements are not reordered if you use an ordered collection.

Examples

```
julia> reduce(*, [2; 3; 4])
24

julia> reduce(*, Int[]; init=1)
1
```

source

`Base.reduce` – Method.

```
reduce(f, A::AbstractArray; dims=:, [init])
```

Reduce 2-argument function `f` along dimensions of `A`. `dims` is a vector specifying the dimensions to reduce, and the keyword argument `init` is the initial value to use in the reductions. For `+`, `*`, `max` and `min` the `init` argument is optional.

The associativity of the reduction is implementation-dependent; if you need a particular associativity, e.g. left-to-right, you should write your own loop or consider using `foldl` or `foldr`. See documentation for `reduce`.

Examples

```
julia> a = reshape(Vector(1:16), (4,4))
4×4 Matrix{Int64}:
 1  5   9  13
 2  6  10  14
 3  7  11  15
 4  8  12  16

julia> reduce(max, a, dims=2)
4×1 Matrix{Int64}:
 13
 14
 15
 16

julia> reduce(max, a, dims=1)
1×4 Matrix{Int64}:
 4  8  12  16
```

source

Base.foldl – Method.

```
foldl(op, itr; [init])
```

Like reduce, but with guaranteed left associativity. If provided, the keyword argument init will be used exactly once. In general, it will be necessary to provide init to work with empty collections.

See also mapfoldl, foldr, accumulate.

Examples

```
julia> foldl(=>, 1:4)
((1 => 2) => 3) => 4

julia> foldl(=>, 1:4; init=0)
(((0 => 1) => 2) => 3) => 4

julia> accumulate(=>, (1,2,3,4))
(1, 1 => 2, (1 => 2) => 3, ((1 => 2) => 3) => 4)
```

source

Base.foldr – Method.

```
foldr(op, itr; [init])
```

Like reduce, but with guaranteed right associativity. If provided, the keyword argument init will be used exactly once. In general, it will be necessary to provide init to work with empty collections.

Examples

```
julia> foldr(=>, 1:4)
1 => (2 => (3 => 4))

julia> foldr(=>, 1:4; init=0)
1 => (2 => (3 => (4 => 0)))
```

source

Base.maximum – Function.

```
maximum(f, A::AbstractArray; dims)
```

Compute the maximum value by calling the function f on each element of an array over the given dimensions.

Examples

```
julia> A = [1 2; 3 4]
2×2 Matrix{Int64}:
 1  2
 3  4
```

```
julia> maximum(abs2, A, dims=1)
1×2 Matrix{Int64}:
 9  16

julia> maximum(abs2, A, dims=2)
2×1 Matrix{Int64}:
  4
 16
```

source

```
maximum(A::AbstractArray; dims)
```

Compute the maximum value of an array over the given dimensions. See also the `max(a,b)` function to take the maximum of two or more arguments, which can be applied elementwise to arrays via `max.(a,b)`.

See also: `maximum!`, `extrema`, `findmax`, `argmax`.

Examples

```
julia> A = [1 2; 3 4]
2×2 Matrix{Int64}:
 1  2
 3  4

julia> maximum(A, dims=1)
1×2 Matrix{Int64}:
 3  4

julia> maximum(A, dims=2)
2×1 Matrix{Int64}:
 2
 4
```

source

```
maximum(itr; [init])
```

Return the largest element in a collection.

The value returned for empty `itr` can be specified by `init`. It must be a neutral element for `max` (i.e. which is less than or equal to any other element) as it is unspecified whether `init` is used for non-empty collections.

Julia 1.6

Keyword argument `init` requires Julia 1.6 or later.

Examples

```
julia> maximum(-20.5:10)
9.5
```

```
julia> maximum([1,2,3])
3

julia> maximum(())
ERROR: ArgumentError: reducing over an empty collection is not allowed; consider supplying
↪  `init` to the reducer
Stacktrace:
[...]

julia> maximum((); init=-Inf)
-Inf
```

source

```
maximum(f, itr; [init])
```

Return the largest result of calling function `f` on each element of `itr`.

The value returned for empty `itr` can be specified by `init`. It must be a neutral element for `max` (i.e. which is less than or equal to any other element) as it is unspecified whether `init` is used for non-empty collections.

Julia 1.6

Keyword argument `init` requires Julia 1.6 or later.

Examples

```
julia> maximum(length, ["Julion", "Julia", "Jule"])
6

julia> maximum(length, []; init=-1)
-1

julia> maximum(sin, Real[]; init=-1.0)  # good, since output of sin is >= -1
-1.0
```

source

`Base.maximum!` – Function.

```
maximum!(r, A)
```

Compute the maximum value of `A` over the singleton dimensions of `r`, and write results to `r`.

Warning

Behavior can be unexpected when any mutated argument shares memory with any other argument.

Examples

```
julia> A = [1 2; 3 4]
2×2 Matrix{Int64}:
 1  2
 3  4

julia> maximum!([1; 1], A)
2-element Vector{Int64}:
 2
 4

julia> maximum!([1 1], A)
1×2 Matrix{Int64}:
 3  4
```

source

Base.minimum – Function.

```
minimum(f, A::AbstractArray; dims)
```

Compute the minimum value by calling the function `f` on each element of an array over the given dimensions.

Examples

```
julia> A = [1 2; 3 4]
2×2 Matrix{Int64}:
 1  2
 3  4

julia> minimum(abs2, A, dims=1)
1×2 Matrix{Int64}:
 1  4

julia> minimum(abs2, A, dims=2)
2×1 Matrix{Int64}:
 1
 9
```

source

```
minimum(A::AbstractArray; dims)
```

Compute the minimum value of an array over the given dimensions. See also the `min(a,b)` function to take the minimum of two or more arguments, which can be applied elementwise to arrays via `min.(a,b)`.

See also: `minimum!`, `extrema`, `findmin`, `argmin`.

Examples

```
julia> A = [1 2; 3 4]
2×2 Matrix{Int64}:
 1  2
```

```
 3  4

julia> minimum(A, dims=1)
1×2 Matrix{Int64}:
 1  2

julia> minimum(A, dims=2)
2×1 Matrix{Int64}:
 1
 3
```

source

```
minimum(itr; [init])
```

Return the smallest element in a collection.

The value returned for empty `itr` can be specified by `init`. It must be a neutral element for `min` (i.e. which is greater than or equal to any other element) as it is unspecified whether `init` is used for non-empty collections.

Julia 1.6

Keyword argument `init` requires Julia 1.6 or later.

Examples

```
julia> minimum(-20.5:10)
-20.5

julia> minimum([1,2,3])
1

julia> minimum([])
ERROR: ArgumentError: reducing over an empty collection is not allowed; consider supplying
↪  `init` to the reducer
Stacktrace:
[...]

julia> minimum([]; init=Inf)
Inf
```

source

```
minimum(f, itr; [init])
```

Return the smallest result of calling function `f` on each element of `itr`.

The value returned for empty `itr` can be specified by `init`. It must be a neutral element for `min` (i.e. which is greater than or equal to any other element) as it is unspecified whether `init` is used for non-empty collections.

Julia 1.6

Keyword argument `init` requires Julia 1.6 or later.

Examples

```
julia> minimum(length, ["Julion", "Julia", "Jule"])
4

julia> minimum(length, []; init=typemax(Int64))
9223372036854775807

julia> minimum(sin, Real[]; init=1.0)  # good, since output of sin is <= 1
1.0
```

source

`Base.minimum!` – Function.

```
minimum!(r, A)
```

Compute the minimum value of `A` over the singleton dimensions of `r`, and write results to `r`.

Warning

Behavior can be unexpected when any mutated argument shares memory with any other argument.

Examples

```
julia> A = [1 2; 3 4]
2×2 Matrix{Int64}:
 1  2
 3  4

julia> minimum!([1; 1], A)
2-element Vector{Int64}:
 1
 3

julia> minimum!([1 1], A)
1×2 Matrix{Int64}:
 1  2
```

source

`Base.extrema` – Function.

```
extrema(f, A::AbstractArray; dims) -> Array{Tuple}
```

Compute the minimum and maximum of `f` applied to each element in the given dimensions of `A`.

Julia 1.2

This method requires Julia 1.2 or later.

source

```
extrema(A::AbstractArray; dims) -> Array{Tuple}
```

Compute the minimum and maximum elements of an array over the given dimensions.

See also: `minimum`, `maximum`, `extrema!`.

Examples

```
julia> A = reshape(Vector(1:2:16), (2,2,2))
2×2×2 Array{Int64, 3}:
[:, :, 1] =
 1  5
 3  7

[:, :, 2] =
  9  13
 11  15

julia> extrema(A, dims = (1,2))
1×1×2 Array{Tuple{Int64, Int64}, 3}:
[:, :, 1] =
 (1, 7)

[:, :, 2] =
 (9, 15)
```

source

```
extrema(f, itr; [init]) -> (mn, mx)
```

Compute both the minimum `mn` and maximum `mx` of `f` applied to each element in `itr` and return them as a 2-tuple. Only one pass is made over `itr`.

The value returned for empty `itr` can be specified by `init`. It must be a 2-tuple whose first and second elements are neutral elements for `min` and `max` respectively (i.e. which are greater/less than or equal to any other element). It is used for non-empty collections. Note: it implies that, for empty `itr`, the returned value `(mn, mx)` satisfies `mn ≥ mx` even though for non-empty `itr` it satisfies `mn ≤ mx`. This is a "paradoxical" but yet expected result.

Julia 1.2

This method requires Julia 1.2 or later.

Julia 1.8

Keyword argument `init` requires Julia 1.8 or later.

Examples

```
julia> extrema(sin, 0:π)
(0.0, 0.9092974268256817)

julia> extrema(sin, Real[]; init = (1.0, -1.0))  # good, since -1 ≤ sin(::Real) ≤ 1
(1.0, -1.0)
```

source

```
extrema(itr; [init]) -> (mn, mx)
```

Compute both the minimum `mn` and maximum `mx` element in a single pass, and return them as a 2-tuple.

The value returned for empty `itr` can be specified by `init`. It must be a 2-tuple whose first and second elements are neutral elements for `min` and `max` respectively (i.e. which are greater/less than or equal to any other element). As a consequence, when `itr` is empty the returned `(mn, mx)` tuple will satisfy `mn ≤ mx`. When `init` is specified it may be used even for non-empty `itr`.

Julia 1.8

Keyword argument `init` requires Julia 1.8 or later.

Examples

```
julia> extrema(2:10)
(2, 10)

julia> extrema([9,pi,4.5])
(3.141592653589793, 9.0)

julia> extrema([]; init = (Inf, -Inf))
(Inf, -Inf)
```

source

`Base.extrema!` – Function.

```
extrema!(r, A)
```

Compute the minimum and maximum value of `A` over the singleton dimensions of `r`, and write results to `r`.

Warning

Behavior can be unexpected when any mutated argument shares memory with any other argument.

Julia 1.8

This method requires Julia 1.8 or later.

Examples

```
julia> A = [1 2; 3 4]
2×2 Matrix{Int64}:
 1  2
 3  4

julia> extrema!([(1, 1); (1, 1)], A)
2-element Vector{Tuple{Int64, Int64}}:
 (1, 2)
 (3, 4)

julia> extrema!([(1, 1);; (1, 1)], A)
1×2 Matrix{Tuple{Int64, Int64}}:
 (1, 3)  (2, 4)
```

source

Base.argmax – Function.

```
argmax(A; dims) -> indices
```

For an array input, return the indices of the maximum elements over the given dimensions. NaN is treated as greater than all other values except missing.

Examples

```
julia> A = [1.0 2; 3 4]
2×2 Matrix{Float64}:
 1.0  2.0
 3.0  4.0

julia> argmax(A, dims=1)
1×2 Matrix{CartesianIndex{2}}:
 CartesianIndex(2, 1)  CartesianIndex(2, 2)

julia> argmax(A, dims=2)
2×1 Matrix{CartesianIndex{2}}:
 CartesianIndex(1, 2)
 CartesianIndex(2, 2)
```

source

```
argmax(itr)
```

Return the index or key of the maximal element in a collection. If there are multiple maximal elements, then the first one will be returned.

The collection must not be empty.

Indices are of the same type as those returned by keys(itr) and pairs(itr).

Values are compared with isless.

See also: argmin, findmax.

Examples

```
julia> argmax([8, 0.1, -9, pi])
1

julia> argmax([1, 7, 7, 6])
2

julia> argmax([1, 7, 7, NaN])
4
```

source

```
argmax(f, domain)
```

Return a value `x` from `domain` for which `f(x)` is maximised. If there are multiple maximal values for `f(x)` then the first one will be found.

`domain` must be a non-empty iterable.

Values are compared with `isless`.

Julia 1.7

This method requires Julia 1.7 or later.

See also `argmin`, `findmax`.

Examples

```
julia> argmax(abs, -10:5)
-10

julia> argmax(cos, 0:π/2:2π)
0.0
```

source

```
argmax(r::AbstractRange)
```

Ranges can have multiple maximal elements. In that case `argmax` will return a maximal index, but not necessarily the first one.

source

`Base.argmin` – Function.

```
argmin(A; dims) -> indices
```

For an array input, return the indices of the minimum elements over the given dimensions. `NaN` is treated as less than all other values except `missing`.

Examples

```
julia> A = [1.0 2; 3 4]
2×2 Matrix{Float64}:
 1.0  2.0
 3.0  4.0

julia> argmin(A, dims=1)
1×2 Matrix{CartesianIndex{2}}:
 CartesianIndex(1, 1)  CartesianIndex(1, 2)

julia> argmin(A, dims=2)
2×1 Matrix{CartesianIndex{2}}:
 CartesianIndex(1, 1)
 CartesianIndex(2, 1)
```

source

```
argmin(itr)
```

Return the index or key of the minimal element in a collection. If there are multiple minimal elements, then the first one will be returned.

The collection must not be empty.

Indices are of the same type as those returned by `keys(itr)` and `pairs(itr)`.

`NaN` is treated as less than all other values except `missing`.

See also: `argmax`, `findmin`.

Examples

```
julia> argmin([8, 0.1, -9, pi])
3

julia> argmin([7, 1, 1, 6])
2

julia> argmin([7, 1, 1, NaN])
4
```

source

```
argmin(f, domain)
```

Return a value `x` from `domain` for which `f(x)` is minimised. If there are multiple minimal values for `f(x)` then the first one will be found.

`domain` must be a non-empty iterable.

`NaN` is treated as less than all other values except `missing`.

Julia 1.7

This method requires Julia 1.7 or later.

See also argmax, findmin.

Examples

```
julia> argmin(sign, -10:5)
-10

julia> argmin(x -> -x^3 + x^2 - 10, -5:5)
5

julia> argmin(acos, 0:0.1:1)
1.0
```

source

```
argmin(r::AbstractRange)
```

Ranges can have multiple minimal elements. In that case argmin will return a minimal index, but not necessarily the first one.

source

Base.findmax – Function.

```
findmax(f, A; dims) -> (f(x), index)
```

For an array input, returns the value in the codomain and index of the corresponding value which maximize f over the given dimensions.

Examples

```
julia> A = [-1.0 1; -0.5 2]
2×2 Matrix{Float64}:
 -1.0  1.0
 -0.5  2.0

julia> findmax(abs2, A, dims=1)
([1.0 4.0], CartesianIndex{2}[CartesianIndex(1, 1) CartesianIndex(2, 2)])

julia> findmax(abs2, A, dims=2)
([1.0; 4.0;;], CartesianIndex{2}[CartesianIndex(1, 1); CartesianIndex(2, 2);;])
```

source

```
findmax(A; dims) -> (maxval, index)
```

For an array input, returns the value and index of the maximum over the given dimensions. NaN is treated as greater than all other values except missing.

Examples

```
julia> A = [1.0 2; 3 4]
2×2 Matrix{Float64}:
 1.0  2.0
```

```
 3.0  4.0

julia> findmax(A, dims=1)
([3.0 4.0], CartesianIndex{2}[CartesianIndex(2, 1) CartesianIndex(2, 2)])

julia> findmax(A, dims=2)
([2.0; 4.0;;], CartesianIndex{2}[CartesianIndex(1, 2); CartesianIndex(2, 2);;])
```

source

```
findmax(itr) -> (x, index)
```

Return the maximal element of the collection `itr` and its index or key. If there are multiple maximal elements, then the first one will be returned. Values are compared with `isless`.

Indices are of the same type as those returned by `keys(itr)` and `pairs(itr)`.

See also: `findmin`, `argmax`, `maximum`.

Examples

```
julia> findmax([8, 0.1, -9, pi])
(8.0, 1)

julia> findmax([1, 7, 7, 6])
(7, 2)

julia> findmax([1, 7, 7, NaN])
(NaN, 4)
```

source

```
findmax(f, domain) -> (f(x), index)
```

Return a pair of a value in the codomain (outputs of `f`) and the index or key of the corresponding value in the `domain` (inputs to `f`) such that `f(x)` is maximised. If there are multiple maximal points, then the first one will be returned.

`domain` must be a non-empty iterable supporting `keys`. Indices are of the same type as those returned by `keys(domain)`.

Values are compared with `isless`.

Julia 1.7

This method requires Julia 1.7 or later.

Examples

```
julia> findmax(identity, 5:9)
(9, 5)

julia> findmax(-, 1:10)
(-1, 1)
```

```
julia> findmax(first, [(1, :a), (3, :b), (3, :c)])
(3, 2)

julia> findmax(cos, 0:π/2:2π)
(1.0, 1)
```

source

Base.findmin – Function.

```
findmin(f, A; dims) -> (f(x), index)
```

For an array input, returns the value in the codomain and index of the corresponding value which minimize `f` over the given dimensions.

Examples

```
julia> A = [-1.0 1; -0.5 2]
2×2 Matrix{Float64}:
 -1.0  1.0
 -0.5  2.0

julia> findmin(abs2, A, dims=1)
([0.25 1.0], CartesianIndex{2}[CartesianIndex(2, 1) CartesianIndex(1, 2)])

julia> findmin(abs2, A, dims=2)
([1.0; 0.25;;], CartesianIndex{2}[CartesianIndex(1, 1); CartesianIndex(2, 1);;])
```

source

```
findmin(A; dims) -> (minval, index)
```

For an array input, returns the value and index of the minimum over the given dimensions. `NaN` is treated as less than all other values except `missing`.

Examples

```
julia> A = [1.0 2; 3 4]
2×2 Matrix{Float64}:
 1.0  2.0
 3.0  4.0

julia> findmin(A, dims=1)
([1.0 2.0], CartesianIndex{2}[CartesianIndex(1, 1) CartesianIndex(1, 2)])

julia> findmin(A, dims=2)
([1.0; 3.0;;], CartesianIndex{2}[CartesianIndex(1, 1); CartesianIndex(2, 1);;])
```

source

```
findmin(itr) -> (x, index)
```

Return the minimal element of the collection `itr` and its index or key. If there are multiple minimal elements, then the first one will be returned. `NaN` is treated as less than all other values except `missing`.

Indices are of the same type as those returned by `keys(itr)` and `pairs(itr)`.

See also: `findmax`, `argmin`, `minimum`.

Examples

```
julia> findmin([8, 0.1, -9, pi])
(-9.0, 3)

julia> findmin([1, 7, 7, 6])
(1, 1)

julia> findmin([1, 7, 7, NaN])
(NaN, 4)
```

source

```
findmin(f, domain) -> (f(x), index)
```

Return a pair of a value in the codomain (outputs of `f`) and the index or key of the corresponding value in the `domain` (inputs to `f`) such that `f(x)` is minimised. If there are multiple minimal points, then the first one will be returned.

`domain` must be a non-empty iterable.

Indices are of the same type as those returned by `keys(domain)` and `pairs(domain)`.

`NaN` is treated as less than all other values except `missing`.

Julia 1.7

This method requires Julia 1.7 or later.

Examples

```
julia> findmin(identity, 5:9)
(5, 1)

julia> findmin(-, 1:10)
(-10, 10)

julia> findmin(first, [(2, :a), (2, :b), (3, :c)])
(2, 1)

julia> findmin(cos, 0:π/2:2π)
(-1.0, 3)
```

source

`Base.findmax!` – Function.

```
findmax!(rval, rind, A) -> (maxval, index)
```

Find the maximum of `A` and the corresponding linear index along singleton dimensions of `rval` and `rind`, and store the results in `rval` and `rind`. `NaN` is treated as greater than all other values except `missing`.

Warning

Behavior can be unexpected when any mutated argument shares memory with any other argument.

source

`Base.findmin!` – Function.

```
findmin!(rval, rind, A) -> (minval, index)
```

Find the minimum of `A` and the corresponding linear index along singleton dimensions of `rval` and `rind`, and store the results in `rval` and `rind`. `NaN` is treated as less than all other values except `missing`.

Warning

Behavior can be unexpected when any mutated argument shares memory with any other argument.

source

`Base.sum` – Function.

```
sum(f, A::AbstractArray; dims)
```

Sum the results of calling function `f` on each element of an array over the given dimensions.

Examples

```
julia> A = [1 2; 3 4]
2×2 Matrix{Int64}:
 1  2
 3  4

julia> sum(abs2, A, dims=1)
1×2 Matrix{Int64}:
 10  20

julia> sum(abs2, A, dims=2)
2×1 Matrix{Int64}:
  5
 25
```

source

```
sum(A::AbstractArray; dims)
```

Sum elements of an array over the given dimensions.

Examples

```
julia> A = [1 2; 3 4]
2×2 Matrix{Int64}:
 1  2
 3  4

julia> sum(A, dims=1)
1×2 Matrix{Int64}:
 4  6

julia> sum(A, dims=2)
2×1 Matrix{Int64}:
 3
 7
```

source

```
sum(itr; [init])
```

Return the sum of all elements in a collection.

The return type is `Int` for signed integers of less than system word size, and `UInt` for unsigned integers of less than system word size. For all other arguments, a common return type is found to which all arguments are promoted.

The value returned for empty `itr` can be specified by `init`. It must be the additive identity (i.e. zero) as it is unspecified whether `init` is used for non-empty collections.

Julia 1.6

Keyword argument `init` requires Julia 1.6 or later.

See also: `reduce`, `mapreduce`, `count`, `union`.

Examples

```
julia> sum(1:20)
210

julia> sum(1:20; init = 0.0)
210.0
```

source

```
sum(f, itr; [init])
```

Sum the results of calling function `f` on each element of `itr`.

The return type is `Int` for signed integers of less than system word size, and `UInt` for unsigned integers of less than system word size. For all other arguments, a common return type is found to which all arguments are promoted.

The value returned for empty `itr` can be specified by `init`. It must be the additive identity (i.e. zero) as it is unspecified whether `init` is used for non-empty collections.

Julia 1.6

Keyword argument `init` requires Julia 1.6 or later.

Examples

```
julia> sum(abs2, [2; 3; 4])
29
```

Note the important difference between `sum(A)` and `reduce(+, A)` for arrays with small integer eltype:

```
julia> sum(Int8[100, 28])
128

julia> reduce(+, Int8[100, 28])
-128
```

In the former case, the integers are widened to system word size and therefore the result is 128. In the latter case, no such widening happens and integer overflow results in -128.

source

`Base.sum!` – Function.

```
sum!(r, A)
```

Sum elements of `A` over the singleton dimensions of `r`, and write results to `r`.

Warning

Behavior can be unexpected when any mutated argument shares memory with any other argument.

Examples

```
julia> A = [1 2; 3 4]
2×2 Matrix{Int64}:
 1  2
 3  4

julia> sum!([1; 1], A)
2-element Vector{Int64}:
 3
 7

julia> sum!([1 1], A)
1×2 Matrix{Int64}:
 4  6
```

source

`Base.prod` – Function.

```
prod(f, A::AbstractArray; dims)
```

Multiply the results of calling the function `f` on each element of an array over the given dimensions.

Examples

```
julia> A = [1 2; 3 4]
2×2 Matrix{Int64}:
 1  2
 3  4

julia> prod(abs2, A, dims=1)
1×2 Matrix{Int64}:
 9  64

julia> prod(abs2, A, dims=2)
2×1 Matrix{Int64}:
   4
 144
```

source

```
prod(A::AbstractArray; dims)
```

Multiply elements of an array over the given dimensions.

Examples

```
julia> A = [1 2; 3 4]
2×2 Matrix{Int64}:
 1  2
 3  4

julia> prod(A, dims=1)
1×2 Matrix{Int64}:
 3  8

julia> prod(A, dims=2)
2×1 Matrix{Int64}:
  2
 12
```

source

```
prod(itr; [init])
```

Return the product of all elements of a collection.

The return type is `Int` for signed integers of less than system word size, and `UInt` for unsigned integers of less than system word size. For all other arguments, a common return type is found to which all arguments are promoted.

The value returned for empty `itr` can be specified by `init`. It must be the multiplicative identity (i.e. one) as it is unspecified whether `init` is used for non-empty collections.

Julia 1.6

Keyword argument `init` requires Julia 1.6 or later.

See also: `reduce`, `cumprod`, `any`.

Examples

```
julia> prod(1:5)
120

julia> prod(1:5; init = 1.0)
120.0
```

source

```
prod(f, itr; [init])
```

Return the product of `f` applied to each element of `itr`.

The return type is `Int` for signed integers of less than system word size, and `UInt` for unsigned integers of less than system word size. For all other arguments, a common return type is found to which all arguments are promoted.

The value returned for empty `itr` can be specified by `init`. It must be the multiplicative identity (i.e. one) as it is unspecified whether `init` is used for non-empty collections.

Julia 1.6

Keyword argument `init` requires Julia 1.6 or later.

Examples

```
julia> prod(abs2, [2; 3; 4])
576
```

source

`Base.prod!` – Function.

```
prod!(r, A)
```

Multiply elements of `A` over the singleton dimensions of `r`, and write results to `r`.

Warning

Behavior can be unexpected when any mutated argument shares memory with any other argument.

Examples

```
julia> A = [1 2; 3 4]
2×2 Matrix{Int64}:
 1  2
 3  4

julia> prod!([1; 1], A)
2-element Vector{Int64}:
  2
 12

julia> prod!([1 1], A)
1×2 Matrix{Int64}:
 3  8
```

source

Base.any – Method.

```
any(itr)::Bool
```

Test whether any elements of a boolean collection are `true`, returning `true` as soon as the first `true` value in `itr` is encountered (short-circuiting). To short-circuit on `false`, use `all`.

If the input contains `missing` values, return `missing` if all non-missing values are `false` (or equivalently, if the input contains no `true` value), following three-valued logic.

See also: `all`, `count`, `sum`, `|`, `||`.

Examples

```
julia> a = [true,false,false,true]
4-element Vector{Bool}:
 1
 0
 0
 1

julia> any(a)
true

julia> any((println(i); v) for (i, v) in enumerate(a))
1
true

julia> any([missing, true])
true

julia> any([false, missing])
missing
```

source

Base.any – Method.

```
any(p, itr)::Bool
```

Determine whether predicate `p` returns `true` for any elements of `itr`, returning `true` as soon as the first item in `itr` for which `p` returns `true` is encountered (short-circuiting). To short-circuit on `false`, use `all`.

If the input contains `missing` values, return `missing` if all non-missing values are `false` (or equivalently, if the input contains no `true` value), following three-valued logic.

Examples

```
julia> any(i->(4<=i<=6), [3,5,7])
true

julia> any(i -> (println(i); i > 3), 1:10)
1
2
3
4
true

julia> any(i -> i > 0, [1, missing])
true

julia> any(i -> i > 0, [-1, missing])
missing

julia> any(i -> i > 0, [-1, 0])
false
```

source

Base.any! – Function.

```
any!(r, A)
```

Test whether any values in `A` along the singleton dimensions of `r` are `true`, and write results to `r`.

Warning

Behavior can be unexpected when any mutated argument shares memory with any other argument.

Examples

```
julia> A = [true false; true false]
2×2 Matrix{Bool}:
 1  0
 1  0

julia> any!(Bool[1; 1], A)
2-element Vector{Bool}:
 1
 1

julia> any!(Bool[1 1], A)
```

```
1×2 Matrix{Bool}:
 1  0
```

source

Base.all – Method.

```
all(itr)::Bool
```

Test whether all elements of a boolean collection are `true`, returning `false` as soon as the first `false` value in `itr` is encountered (short-circuiting). To short-circuit on `true`, use any.

If the input contains `missing` values, return `missing` if all non-missing values are `true` (or equivalently, if the input contains no `false` value), following three-valued logic.

See also: `all!`, any, `count`, `&`, `&&`, `allunique`.

Examples

```
julia> a = [true,false,false,true]
4-element Vector{Bool}:
 1
 0
 0
 1

julia> all(a)
false

julia> all((println(i); v) for (i, v) in enumerate(a))
1
2
false

julia> all([missing, false])
false

julia> all([true, missing])
missing
```

source

Base.all – Method.

```
all(p, itr)::Bool
```

Determine whether predicate `p` returns `true` for all elements of `itr`, returning `false` as soon as the first item in `itr` for which `p` returns `false` is encountered (short-circuiting). To short-circuit on `true`, use any.

If the input contains `missing` values, return `missing` if all non-missing values are `true` (or equivalently, if the input contains no `false` value), following three-valued logic.

Examples

```
julia> all(i->(4<=i<=6), [4,5,6])
true

julia> all(i -> (println(i); i < 3), 1:10)
1
2
3
false

julia> all(i -> i > 0, [1, missing])
missing

julia> all(i -> i > 0, [-1, missing])
false

julia> all(i -> i > 0, [1, 2])
true
```

source

Base.all! – Function.

```
all!(r, A)
```

Test whether all values in A along the singleton dimensions of r are true, and write results to r.

Warning

Behavior can be unexpected when any mutated argument shares memory with any other argument.

Examples

```
julia> A = [true false; true false]
2×2 Matrix{Bool}:
 1  0
 1  0

julia> all!(Bool[1; 1], A)
2-element Vector{Bool}:
 0
 0

julia> all!(Bool[1 1], A)
1×2 Matrix{Bool}:
 1  0
```

source

Base.count – Function.

```
count([f=identity,] A::AbstractArray; dims=:)
```

Count the number of elements in `A` for which `f` returns `true` over the given dimensions.

Julia 1.5

`dims` keyword was added in Julia 1.5.

Julia 1.6

`init` keyword was added in Julia 1.6.

Examples

```
julia> A = [1 2; 3 4]
2×2 Matrix{Int64}:
 1  2
 3  4

julia> count(<=(2), A, dims=1)
1×2 Matrix{Int64}:
 1  1

julia> count(<=(2), A, dims=2)
2×1 Matrix{Int64}:
 2
 0
```

source

```
count(
    pattern::Union{AbstractChar,AbstractString,AbstractPattern},
    string::AbstractString;
    overlap::Bool = false,
)
```

Return the number of matches for `pattern` in `string`. This is equivalent to calling `length(findall(pattern, string))` but more efficient.

If `overlap=true`, the matching sequences are allowed to overlap indices in the original string, otherwise they must be from disjoint character ranges.

Julia 1.3

This method requires at least Julia 1.3.

Julia 1.7

Using a character as the pattern requires at least Julia 1.7.

Examples

```
julia> count('a', "JuliaLang")
2
```

```
julia> count(r"a(.)a", "cabacabac", overlap=true)
3

julia> count(r"a(.)a", "cabacabac")
2
```

See also

eachmatch, occursin, findall

source

```
count([f=identity,] itr; init=0)::Integer
```

Count the number of elements in `itr` for which the function `f` returns `true`. If `f` is omitted, count the number of `true` elements in `itr` (which should be a collection of boolean values). `init` optionally specifies the value to start counting from and therefore also determines the output type.

Julia 1.6

`init` keyword was added in Julia 1.6.

See also: any, sum.

Examples

```
julia> count(i->(4<=i<=6), [2,3,4,5,6])
3

julia> count([true, false, true, true])
3

julia> count(>(3), 1:7, init=UInt(0))
0x0000000000000004
```

source

Base.foreach – Function.

```
foreach(f, c...) -> nothing
```

Call function `f` on each element of iterable `c`. For multiple iterable arguments, `f` is called elementwise, and iteration stops when any iterator is finished.

`foreach` should be used instead of map when the results of `f` are not needed, for example in `foreach(println, array)`.

Examples

```
julia> tri = 1:3:7; res = Int[];

julia> foreach(x -> push!(res, x^2), tri)

julia> res
```

```
3-element Vector{Int64}:
  1
 16
 49

julia> foreach((x, y) -> println(x, " with ", y), tri, 'a':'z')
1 with a
4 with b
7 with c
```

source

Base.map – Function.

```
map(f, A::AbstractArray...) -> N-array
```

When acting on multi-dimensional arrays of the same ndims, they must all have the same axes, and the answer will too.

See also broadcast, which allows mismatched sizes.

Examples

```
julia> map(//, [1 2; 3 4], [4 3; 2 1])
2×2 Matrix{Rational{Int64}}:
 1//4  2//3
 3//2  4//1

julia> map(+, [1 2; 3 4], zeros(2,1))
ERROR: DimensionMismatch

julia> map(+, [1 2; 3 4], [1,10,100,1000], zeros(3,1))  # iterates until 3rd is exhausted
3-element Vector{Float64}:
   2.0
  13.0
 102.0
```

source

```
map(f, c...) -> collection
```

Transform collection c by applying f to each element. For multiple collection arguments, apply f elementwise, and stop when any of them is exhausted.

The element type of the result is determined in the same manner as in collect.

See also map!, foreach, mapreduce, mapslices, zip, Iterators.map.

Examples

```
julia> map(x -> x * 2, [1, 2, 3])
3-element Vector{Int64}:
 2
 4
 6
```

```
julia> map(+, [1, 2, 3], [10, 20, 30, 400, 5000])
3-element Vector{Int64}:
 11
 22
 33
```

source

Base.map! – Function.

```
map!(f, values(dict::AbstractDict))
```

Modifies `dict` by transforming each value from `val` to `f(val)`. Note that the type of `dict` cannot be changed: if `f(val)` is not an instance of the value type of `dict` then it will be converted to the value type if possible and otherwise raise an error.

Julia 1.2

`map!(f, values(dict::AbstractDict))` requires Julia 1.2 or later.

Examples

```
julia> d = Dict(:a => 1, :b => 2)
Dict{Symbol, Int64} with 2 entries:
  :a => 1
  :b => 2

julia> map!(v -> v-1, values(d))
ValueIterator for a Dict{Symbol, Int64} with 2 entries. Values:
  0
  1
```

source

```
map!(function, array)
```

Like `map`, but stores the result in the same array.

Julia 1.12

This method requires Julia 1.12 or later. To support previous versions too, use the equivalent `map!(function, array, array)`.

Examples

```
julia> a = [1 2 3; 4 5 6];

julia> map!(x -> x^3, a);

julia> a
2×3 Matrix{Int64}:
  1    8   27
 64  125  216
```

source

```
map!(function, destination, collection...)
```

Like `map`, but stores the result in `destination` rather than a new collection. `destination` must be at least as large as the smallest collection.

Warning

Behavior can be unexpected when any mutated argument shares memory with any other argument.

See also: `map`, `foreach`, `zip`, `copyto!`.

Examples

```
julia> a = zeros(3);

julia> map!(x -> x * 2, a, [1, 2, 3]);

julia> a
3-element Vector{Float64}:
 2.0
 4.0
 6.0

julia> map!(+, zeros(Int, 5), 100:999, 1:3)
5-element Vector{Int64}:
 101
 103
 105
   0
   0
```

source

`Base.mapreduce` – Method.

```
mapreduce(f, op, itrs...; [init])
```

Apply function `f` to each element(s) in `itrs`, and then reduce the result using the binary function `op`. If provided, `init` must be a neutral element for `op` that will be returned for empty collections. It is unspecified whether `init` is used for non-empty collections. In general, it will be necessary to provide `init` to work with empty collections.

`mapreduce` is functionally equivalent to calling `reduce(op, map(f, itr); init=init)`, but will in general execute faster since no intermediate collection needs to be created. See documentation for `reduce` and `map`.

Julia 1.2

`mapreduce` with multiple iterators requires Julia 1.2 or later.

Examples

```
julia> mapreduce(x->x^2, +, [1:3;]) # == 1 + 4 + 9
14
```

The associativity of the reduction is implementation-dependent. Additionally, some implementations may reuse the return value of `f` for elements that appear multiple times in `itr`. Use `mapfoldl` or `mapfoldr` instead for guaranteed left or right associativity and invocation of `f` for every value.

source

`Base.mapfoldl` – Method.

```
mapfoldl(f, op, itr; [init])
```

Like `mapreduce`, but with guaranteed left associativity, as in `foldl`. If provided, the keyword argument `init` will be used exactly once. In general, it will be necessary to provide `init` to work with empty collections.

source

`Base.mapfoldr` – Method.

```
mapfoldr(f, op, itr; [init])
```

Like `mapreduce`, but with guaranteed right associativity, as in `foldr`. If provided, the keyword argument `init` will be used exactly once. In general, it will be necessary to provide `init` to work with empty collections.

source

`Base.first` – Function.

```
first(s::AbstractString, n::Integer)
```

Get a string consisting of the first `n` characters of `s`.

Examples

```
julia> first("∀ϵ≠0: ϵ²>0", 0)
""

julia> first("∀ϵ≠0: ϵ²>0", 1)
"∀"

julia> first("∀ϵ≠0: ϵ²>0", 3)
"∀ϵ≠"
```

source

```
first(itr, n::Integer)
```

Get the first `n` elements of the iterable collection `itr`, or fewer elements if `itr` is not long enough.

See also: `startswith`, `Iterators.take`.

Julia 1.6

This method requires at least Julia 1.6.

Examples

```
julia> first(["foo", "bar", "qux"], 2)
2-element Vector{String}:
 "foo"
 "bar"

julia> first(1:6, 10)
1:6

julia> first(Bool[], 1)
Bool[]
```

source

```
first(coll)
```

Get the first element of an iterable collection. Return the start point of an `AbstractRange` even if it is empty.

See also: `only`, `firstindex`, `last`.

Examples

```
julia> first(2:2:10)
2

julia> first([1; 2; 3; 4])
1
```

source

`Base.last` – Function.

```
last(s::AbstractString, n::Integer)
```

Get a string consisting of the last `n` characters of `s`.

Examples

```
julia> last("∀ϵ≠0: ϵ²>0", 0)
""

julia> last("∀ϵ≠0: ϵ²>0", 1)
"0"

julia> last("∀ϵ≠0: ϵ²>0", 3)
"²>0"
```

source

```
last(itr, n::Integer)
```

Get the last n elements of the iterable collection `itr`, or fewer elements if `itr` is not long enough.

Julia 1.6

This method requires at least Julia 1.6.

Examples

```
julia> last(["foo", "bar", "qux"], 2)
2-element Vector{String}:
 "bar"
 "qux"

julia> last(1:6, 10)
1:6

julia> last(Float64[], 1)
Float64[]
```

source

```
last(coll)
```

Get the last element of an ordered collection, if it can be computed in O(1) time. This is accomplished by calling `lastindex` to get the last index. Return the end point of an `AbstractRange` even if it is empty.

See also `first`, `endswith`.

Examples

```
julia> last(1:2:10)
9

julia> last([1; 2; 3; 4])
4
```

source

`Base.front` – Function.

```
front(x::Tuple)::Tuple
```

Return a `Tuple` consisting of all but the last component of `x`.

See also: `first`, `tail`.

Examples

```
julia> Base.front((1,2,3))
(1, 2)

julia> Base.front(())
ERROR: ArgumentError: Cannot call front on an empty tuple.
```

source

Base.tail – Function.

```
tail(x::Tuple)::Tuple
```

Return a Tuple consisting of all but the first component of x.

See also: front, rest, first, Iterators.peel.

Examples

```
julia> Base.tail((1,2,3))
(2, 3)

julia> Base.tail(())
ERROR: ArgumentError: Cannot call tail on an empty tuple.
```

source

Base.step – Function.

```
step(r)
```

Get the step size of an AbstractRange object.

Examples

```
julia> step(1:10)
1

julia> step(1:2:10)
2

julia> step(2.5:0.3:10.9)
0.3

julia> step(range(2.5, stop=10.9, length=85))
0.1
```

source

Base.collect – Method.

```
collect(iterator)
```

Return an Array of all items in a collection or iterator. For dictionaries, returns a Vector of key=>value Pairs. If the argument is array-like or is an iterator with the HasShape trait, the result will have the same shape and number of dimensions as the argument.

Used by comprehensions to turn a generator expression into an Array. Thus, *on generators*, the square-brackets notation may be used instead of calling collect, see second example.

The element type of the returned array is based on the types of the values collected. However, if the iterator is empty then the element type of the returned (empty) array is determined by type inference.

Examples

Collect items from a UnitRange{Int64} collection:

```
julia> collect(1:3)
3-element Vector{Int64}:
 1
 2
 3
```

Collect items from a generator (same output as `[x^2 for x in 1:3]`):

```
julia> collect(x^2 for x in 1:3)
3-element Vector{Int64}:
 1
 4
 9
```

Collecting an empty iterator where the result type depends on type inference:

```
julia> [rand(Bool) ? 1 : missing for _ in []]
Union{Missing, Int64}[]
```

When the iterator is non-empty, the result type depends only on values:

```
julia> [rand(Bool) ? 1 : missing for _ in [""]]
1-element Vector{Int64}:
 1
```

source

`Base.collect` – Method.

```
collect(element_type, collection)
```

Return an `Array` with the given element type of all items in a collection or iterable. The result has the same shape and number of dimensions as `collection`.

Examples

```
julia> collect(Float64, 1:2:5)
3-element Vector{Float64}:
 1.0
 3.0
 5.0
```

source

`Base.filter` – Function.

```
filter(f, itr::SkipMissing{<:AbstractArray})
```

Return a vector similar to the array wrapped by the given `SkipMissing` iterator but with all missing elements and those for which `f` returns `false` removed.

Julia 1.2

This method requires Julia 1.2 or later.

Examples

```
julia> x = [1 2; missing 4]
2×2 Matrix{Union{Missing, Int64}}:
 1         2
  missing  4

julia> filter(isodd, skipmissing(x))
1-element Vector{Int64}:
 1
```

source

```
filter(f, d::AbstractDict)
```

Return a copy of `d`, removing elements for which `f` is `false`. The function `f` is passed `key=>value` pairs.

Examples

```
julia> d = Dict(1=>"a", 2=>"b")
Dict{Int64, String} with 2 entries:
  2 => "b"
  1 => "a"

julia> filter(p->isodd(p.first), d)
Dict{Int64, String} with 1 entry:
  1 => "a"
```

source

```
filter(f)
```

Create a function that filters its arguments with function `f` using `filter`, i.e. a function equivalent to `x -> filter(f, x)`.

The returned function is of type `Base.Fix1{typeof(filter)}`, which can be used to implement specialized methods.

Examples

```
julia> (1, 2, Inf, 4, NaN, 6) |> filter(isfinite)
(1, 2, 4, 6)

julia> map(filter(iseven), [1:3, 2:4, 3:5])
3-element Vector{Vector{Int64}}:
 [2]
 [2, 4]
 [4]
```

Julia 1.9

This method requires at least Julia 1.9.

source

```
filter(f, a)
```

Return a copy of collection a, removing elements for which f is false. The function f is passed one argument.

Julia 1.4

Support for a as a tuple requires at least Julia 1.4.

See also: filter!, Iterators.filter.

Examples

```
julia> a = 1:10
1:10

julia> filter(isodd, a)
5-element Vector{Int64}:
 1
 3
 5
 7
 9
```

source

Base.filter! – Function.

```
filter!(f, d::AbstractDict)
```

Update d, removing elements for which f is false. The function f is passed key=>value pairs.

Examples

```
julia> d = Dict(1=>"a", 2=>"b", 3=>"c")
Dict{Int64, String} with 3 entries:
  2 => "b"
  3 => "c"
  1 => "a"

julia> filter!(p->isodd(p.first), d)
Dict{Int64, String} with 2 entries:
  3 => "c"
  1 => "a"
```

source

```
filter!(f, a)
```

Update collection a, removing elements for which f is false. The function f is passed one argument.

Examples

```
julia> filter!(isodd, Vector(1:10))
5-element Vector{Int64}:
 1
 3
 5
 7
 9
```

source

Base.replace – Method.

```
replace(A, old_new::Pair...; [count::Integer])
```

Return a copy of collection A where, for each pair old=>new in old_new, all occurrences of old are replaced by new. Equality is determined using isequal. If count is specified, then replace at most count occurrences in total.

The element type of the result is chosen using promotion (see promote_type) based on the element type of A and on the types of the new values in pairs. If count is omitted and the element type of A is a Union, the element type of the result will not include singleton types which are replaced with values of a different type: for example, Union{T,Missing} will become T if missing is replaced.

See also replace!, splice!, delete!, insert!.

Julia 1.7

Version 1.7 is required to replace elements of a Tuple.

Examples

```
julia> replace([1, 2, 1, 3], 1=>0, 2=>4, count=2)
4-element Vector{Int64}:
 0
 4
 1
 3

julia> replace([1, missing], missing=>0)
2-element Vector{Int64}:
 1
 0
```

source

Base.replace – Method.

```
replace(new::Union{Function, Type}, A; [count::Integer])
```

Return a copy of A where each value x in A is replaced by new(x). If count is specified, then replace at most count values in total (replacements being defined as new(x) !== x).

Julia 1.7

Version 1.7 is required to replace elements of a Tuple.

Examples

```
julia> replace(x -> isodd(x) ? 2x : x, [1, 2, 3, 4])
4-element Vector{Int64}:
 2
 2
 6
 4

julia> replace(Dict(1=>2, 3=>4)) do kv
           first(kv) < 3 ? first(kv)=>3 : kv
       end
Dict{Int64, Int64} with 2 entries:
  3 => 4
  1 => 3
```

source

Base.replace! – Function.

```
replace!(new::Union{Function, Type}, A; [count::Integer])
```

Replace each element `x` in collection `A` by `new(x)`. If `count` is specified, then replace at most `count` values in total (replacements being defined as `new(x) !== x`).

Examples

```
julia> replace!(x -> isodd(x) ? 2x : x, [1, 2, 3, 4])
4-element Vector{Int64}:
 2
 2
 6
 4

julia> replace!(Dict(1=>2, 3=>4)) do kv
           first(kv) < 3 ? first(kv)=>3 : kv
       end
Dict{Int64, Int64} with 2 entries:
  3 => 4
  1 => 3

julia> replace!(x->2x, Set([3, 6]))
Set{Int64} with 2 elements:
  6
  12
```

source

```
replace!(A, old_new::Pair...; [count::Integer])
```

For each pair `old=>new` in `old_new`, replace all occurrences of `old` in collection `A` by `new`. Equality is determined using `isequal`. If `count` is specified, then replace at most `count` occurrences in total. See also `replace`.

Examples

```
julia> replace!([1, 2, 1, 3], 1=>0, 2=>4, count=2)
4-element Vector{Int64}:
 0
 4
 1
 3

julia> replace!(Set([1, 2, 3]), 1=>0)
Set{Int64} with 3 elements:
  0
  2
  3
```

source

Base.rest – Function.

```
Base.rest(collection[, itr_state])
```

Generic function for taking the tail of `collection`, starting from a specific iteration state `itr_state`. Return a `Tuple`, if `collection` itself is a `Tuple`, a subtype of `AbstractVector`, if `collection` is an `AbstractArray`, a subtype of `AbstractString` if `collection` is an `AbstractString`, and an arbitrary iterator, falling back to `Iterators.rest(collection[, itr_state])`, otherwise.

Can be overloaded for user-defined collection types to customize the behavior of slurping in assignments in final position, like `a, b... = collection`.

Julia 1.6

`Base.rest` requires at least Julia 1.6.

See also: `first`, `Iterators.rest`, `Base.split_rest`.

Examples

```
julia> a = [1 2; 3 4]
2×2 Matrix{Int64}:
 1  2
 3  4

julia> first, state = iterate(a)
(1, 2)

julia> first, Base.rest(a, state)
(1, [3, 2, 4])
```

source

Base.split_rest – Function.

```
Base.split_rest(collection, n::Int[, itr_state]) -> (rest_but_n, last_n)
```

Generic function for splitting the tail of `collection`, starting from a specific iteration state `itr_state`. Returns a tuple of two new collections. The first one contains all elements of the tail but the `n` last ones, which make up the second collection.

The type of the first collection generally follows that of `Base.rest`, except that the fallback case is not lazy, but is collected eagerly into a vector.

Can be overloaded for user-defined collection types to customize the behavior of slurping in assignments in non-final position, like `a, b..., c = collection`.

Julia 1.9

`Base.split_rest` requires at least Julia 1.9.

See also: `Base.rest`.

Examples

```
julia> a = [1 2; 3 4]
2×2 Matrix{Int64}:
 1  2
 3  4

julia> first, state = iterate(a)
(1, 2)

julia> first, Base.split_rest(a, 1, state)
(1, ([3, 2], [4]))
```

source

45.5 Indexable Collections

`Base.getindex` – Function.

```
getindex(collection, key...)
```

Retrieve the value(s) stored at the given key or index within a collection. The syntax `a[i,j,...]` is converted by the compiler to `getindex(a, i, j, ...)`.

See also `get`, `keys`, `eachindex`.

Examples

```
julia> A = Dict("a" => 1, "b" => 2)
Dict{String, Int64} with 2 entries:
  "b" => 2
  "a" => 1

julia> getindex(A, "a")
1
```

source

`Base.setindex!` – Function.

```
setindex!(collection, value, key...)
```

Store the given value at the given key or index within a collection. The syntax `a[i,j,...] = x` is converted by the compiler to `(setindex!(a, x, i, j, ...); x)`.

Examples

```
julia> a = Dict("a"=>1)
Dict{String, Int64} with 1 entry:
  "a" => 1

julia> setindex!(a, 2, "b")
Dict{String, Int64} with 2 entries:
  "b" => 2
  "a" => 1
```

source

Base.firstindex – Function.

```
firstindex(collection)::Integer
firstindex(collection, d)::Integer
```

Return the first index of `collection`. If d is given, return the first index of `collection` along dimension d.

The syntaxes `A[begin]` and `A[1, begin]` lower to `A[firstindex(A)]` and `A[1, firstindex(A, 2)]`, respectively.

See also: `first`, `axes`, `lastindex`, `nextind`.

Examples

```
julia> firstindex([1,2,4])
1

julia> firstindex(rand(3,4,5), 2)
1
```

source

Base.lastindex – Function.

```
lastindex(collection)::Integer
lastindex(collection, d)::Integer
```

Return the last index of `collection`. If d is given, return the last index of `collection` along dimension d.

The syntaxes `A[end]` and `A[end, end]` lower to `A[lastindex(A)]` and `A[lastindex(A, 1), lastindex(A, 2)]`, respectively.

See also: `axes`, `firstindex`, `eachindex`, `prevind`.

Examples

```
julia> lastindex([1,2,4])
3

julia> lastindex(rand(3,4,5), 2)
4
```

source

Fully implemented by:

- Array
- BitArray
- AbstractArray
- SubArray

Partially implemented by:

- AbstractRange
- UnitRange
- Tuple
- AbstractString
- Dict
- IdDict
- WeakKeyDict
- NamedTuple

45.6 Dictionaries

Dict is the standard dictionary. Its implementation uses hash as the hashing function for the key, and isequal to determine equality. Define these two functions for custom types to override how they are stored in a hash table.

IdDict is a special hash table where the keys are always object identities.

WeakKeyDict is a hash table implementation where the keys are weak references to objects, and thus may be garbage collected even when referenced in a hash table. Like Dict it uses hash for hashing and isequal for equality, unlike Dict it does not convert keys on insertion.

Dicts can be created by passing pair objects constructed with => to a Dict constructor: Dict("A"=>1, "B"=>2). This call will attempt to infer type information from the keys and values (i.e. this example creates a Dict{String, Int64}). To explicitly specify types use the syntax Dict{KeyType,ValueType}(...). For example, Dict{String,Int32}("A"=>1, "B"=>2).

Dictionaries may also be created with generators. For example, Dict(i => f(i) for i = 1:10).

Given a dictionary `D`, the syntax `D[x]` returns the value of key `x` (if it exists) or throws an error, and `D[x] = y` stores the key-value pair `x => y` in `D` (replacing any existing value for the key `x`). Multiple arguments to `D[...]` are converted to tuples; for example, the syntax `D[x,y]` is equivalent to `D[(x,y)]`, i.e. it refers to the value keyed by the tuple `(x,y)`.

`Base.AbstractDict` – Type.

```
AbstractDict{K, V}
```

Supertype for dictionary-like types with keys of type `K` and values of type `V`. `Dict`, `IdDict` and other types are subtypes of this. An `AbstractDict{K, V}` should be an iterator of `Pair{K, V}`.

source

`Base.Dict` – Type.

```
Dict([itr])
```

`Dict{K,V}()` constructs a hash table with keys of type `K` and values of type `V`. Keys are compared with `isequal` and hashed with `hash`.

Given a single iterable argument, constructs a `Dict` whose key-value pairs are taken from 2-tuples `(key,value)` generated by the argument.

Examples

```
julia> Dict([("A", 1), ("B", 2)])
Dict{String, Int64} with 2 entries:
  "B" => 2
  "A" => 1
```

Alternatively, a sequence of pair arguments may be passed.

```
julia> Dict("A"=>1, "B"=>2)
Dict{String, Int64} with 2 entries:
  "B" => 2
  "A" => 1
```

Warning

Keys are allowed to be mutable, but if you do mutate stored keys, the hash table may become internally inconsistent, in which case the `Dict` will not work properly. `IdDict` can be an alternative if you need to mutate keys.

source

`Base.IdDict` – Type.

```
IdDict([itr])
```

`IdDict{K,V}()` constructs a hash table using `objectid` as hash and `===` as equality with keys of type `K` and values of type `V`. See `Dict` for further help and `IdSet` for the set version of this.

In the example below, the `Dict` keys are all `isequal` and therefore get hashed the same, so they get overwritten. The `IdDict` hashes by object-id, and thus preserves the 3 different keys.

Examples

```
julia> Dict(true => "yes", 1 => "no", 1.0 => "maybe")
Dict{Real, String} with 1 entry:
  1.0 => "maybe"

julia> IdDict(true => "yes", 1 => "no", 1.0 => "maybe")
IdDict{Any, String} with 3 entries:
  true => "yes"
  1.0  => "maybe"
  1    => "no"
```

source

Base.WeakKeyDict – Type.

```
WeakKeyDict([itr])
```

WeakKeyDict() constructs a hash table where the keys are weak references to objects which may be garbage collected even when referenced in a hash table.

See Dict for further help. Note, unlike Dict, WeakKeyDict does not convert keys on insertion, as this would imply the key object was unreferenced anywhere before insertion.

See also WeakRef.

source

Base.ImmutableDict – Type.

```
ImmutableDict
```

ImmutableDict is a dictionary implemented as an immutable linked list, which is optimal for small dictionaries that are constructed over many individual insertions. Note that it is not possible to remove a value, although it can be partially overridden and hidden by inserting a new value with the same key.

```
ImmutableDict(KV::Pair)
```

Create a new entry in the ImmutableDict for a key => value pair

- use (key => value) in dict to see if this particular combination is in the properties set
- use get(dict, key, default) to retrieve the most recent value for a particular key

source

Base.PersistentDict – Type.

```
PersistentDict
```

PersistentDict is a dictionary implemented as a hash array mapped trie, which is optimal for situations where you need persistence, each operation returns a new dictionary separate from the previous one, but the underlying implementation is space-efficient and may share storage across multiple separate dictionaries.

Note

It behaves like an IdDict.

```
PersistentDict(KV::Pair)
```

Examples

```
julia> dict = Base.PersistentDict(:a=>1)
Base.PersistentDict{Symbol, Int64} with 1 entry:
  :a => 1

julia> dict2 = Base.delete(dict, :a)
Base.PersistentDict{Symbol, Int64}()

julia> dict3 = Base.PersistentDict(dict, :a=>2)
Base.PersistentDict{Symbol, Int64} with 1 entry:
  :a => 2
```

source

Base.haskey – Function.

```
haskey(collection, key)::Bool
```

Determine whether a collection has a mapping for a given key.

Examples

```
julia> D = Dict('a'=>2, 'b'=>3)
Dict{Char, Int64} with 2 entries:
  'a' => 2
  'b' => 3

julia> haskey(D, 'a')
true

julia> haskey(D, 'c')
false
```

source

Base.get – Function.

```
get(f::Union{Function, Type}, collection, key)
```

Return the value stored for the given key, or if no mapping for the key is present, return `f()`. Use `get!` to also store the default value in the dictionary.

This is intended to be called using do block syntax

```
get(dict, key) do
    # default value calculated here
    time()
end
```

source

```
get(collection, key, default)
```

Return the value stored for the given key, or the given default value if no mapping for the key is present.

Julia 1.7

For tuples and numbers, this function requires at least Julia 1.7.

Examples

```
julia> d = Dict("a"=>1, "b"=>2);

julia> get(d, "a", 3)
1

julia> get(d, "c", 3)
3
```

source

Base.get! – Function.

```
get!(f::Union{Function, Type}, collection, key)
```

Return the value stored for the given key, or if no mapping for the key is present, store `key => f()`, and return `f()`.

This is intended to be called using do block syntax.

Examples

```
julia> squares = Dict{Int, Int}();

julia> function get_square!(d, i)
           get!(d, i) do
               i^2
           end
       end
get_square! (generic function with 1 method)

julia> get_square!(squares, 2)
4

julia> squares
Dict{Int64, Int64} with 1 entry:
  2 => 4
```

source

```
get!(collection, key, default)
```

Return the value stored for the given key, or if no mapping for the key is present, store `key => default`, and return `default`.

Examples

```
julia> d = Dict("a"=>1, "b"=>2, "c"=>3);

julia> get!(d, "a", 5)
1

julia> get!(d, "d", 4)
4

julia> d
Dict{String, Int64} with 4 entries:
  "c" => 3
  "b" => 2
  "a" => 1
  "d" => 4
```

source

Base.getkey – Function.

```
getkey(collection, key, default)
```

Return the key matching argument key if one exists in `collection`, otherwise return `default`.

Examples

```
julia> D = Dict('a'=>2, 'b'=>3)
Dict{Char, Int64} with 2 entries:
  'a' => 2
  'b' => 3

julia> getkey(D, 'a', 1)
'a': ASCII/Unicode U+0061 (category Ll: Letter, lowercase)

julia> getkey(D, 'd', 'a')
'a': ASCII/Unicode U+0061 (category Ll: Letter, lowercase)
```

source

Base.delete! – Function.

```
delete!(collection, key)
```

Delete the mapping for the given key in a collection, if any, and return the collection.

Examples

```
julia> d = Dict("a"=>1, "b"=>2)
Dict{String, Int64} with 2 entries:
  "b" => 2
  "a" => 1

julia> delete!(d, "b")
Dict{String, Int64} with 1 entry:
  "a" => 1
```

```
julia> delete!(d, "b") # d is left unchanged
Dict{String, Int64} with 1 entry:
  "a" => 1
```

source

Base.pop! – Method.

```
pop!(collection, key[, default])
```

Delete and return the mapping for key if it exists in collection, otherwise return default, or throw an error if default is not specified.

Examples

```
julia> d = Dict("a"=>1, "b"=>2, "c"=>3);

julia> pop!(d, "a")
1

julia> pop!(d, "d")
ERROR: KeyError: key "d" not found
Stacktrace:
[...]

julia> pop!(d, "e", 4)
4
```

source

Base.keys – Function.

```
keys(iterator)
```

For an iterator or collection that has keys and values (e.g. arrays and dictionaries), return an iterator over the keys.

source

Base.values – Function.

```
values(a::AbstractDict)
```

Return an iterator over all values in a collection. collect(values(a)) returns an array of values. When the values are stored internally in a hash table, as is the case for Dict, the order in which they are returned may vary. But keys(a), values(a) and pairs(a) all iterate a and return the elements in the same order.

Examples

```
julia> D = Dict('a'=>2, 'b'=>3)
Dict{Char, Int64} with 2 entries:
  'a' => 2
  'b' => 3
```

```
julia> collect(values(D))
2-element Vector{Int64}:
 2
 3
```

source

```
values(iterator)
```

For an iterator or collection that has keys and values, return an iterator over the values. This function simply returns its argument by default, since the elements of a general iterator are normally considered its "values".

Examples

```
julia> d = Dict("a"=>1, "b"=>2);

julia> values(d)
ValueIterator for a Dict{String, Int64} with 2 entries. Values:
  2
  1

julia> values([2])
1-element Vector{Int64}:
 2
```

source

Base.pairs – Function.

```
pairs(collection)
```

Return an iterator over `key => value` pairs for any collection that maps a set of keys to a set of values. This includes arrays, where the keys are the array indices. When the entries are stored internally in a hash table, as is the case for `Dict`, the order in which they are returned may vary. But `keys(a)`, `values(a)` and `pairs(a)` all iterate a and return the elements in the same order.

Examples

```
julia> a = Dict(zip(["a", "b", "c"], [1, 2, 3]))
Dict{String, Int64} with 3 entries:
  "c" => 3
  "b" => 2
  "a" => 1

julia> pairs(a)
Dict{String, Int64} with 3 entries:
  "c" => 3
  "b" => 2
  "a" => 1

julia> foreach(println, pairs(["a", "b", "c"]))
```

```
1 => "a"
2 => "b"
3 => "c"

julia> (;a=1, b=2, c=3) |> pairs |> collect
3-element Vector{Pair{Symbol, Int64}}:
 :a => 1
 :b => 2
 :c => 3

julia> (;a=1, b=2, c=3) |> collect
3-element Vector{Int64}:
 1
 2
 3
```

source

```
pairs(IndexLinear(), A)
pairs(IndexCartesian(), A)
pairs(IndexStyle(A), A)
```

An iterator that accesses each element of the array `A`, returning `i => x`, where `i` is the index for the element and `x = A[i]`. Identical to `pairs(A)`, except that the style of index can be selected. Also similar to `enumerate(A)`, except `i` will be a valid index for `A`, while `enumerate` always counts from 1 regardless of the indices of `A`.

Specifying `IndexLinear()` ensures that `i` will be an integer; specifying `IndexCartesian()` ensures that `i` will be a `Base.CartesianIndex`; specifying `IndexStyle(A)` chooses whichever has been defined as the native indexing style for array `A`.

Mutation of the bounds of the underlying array will invalidate this iterator.

Examples

```
julia> A = ["a" "d"; "b" "e"; "c" "f"];

julia> for (index, value) in pairs(IndexStyle(A), A)
           println("$index $value")
       end
1 a
2 b
3 c
4 d
5 e
6 f

julia> S = view(A, 1:2, :);

julia> for (index, value) in pairs(IndexStyle(S), S)
           println("$index $value")
       end
CartesianIndex(1, 1) a
CartesianIndex(2, 1) b
```

```
CartesianIndex(1, 2) d
CartesianIndex(2, 2) e
```

See also IndexStyle, axes.

source

Base.merge – Function.

```
merge(initial::StyledStrings.Face, others::StyledStrings.Face...)
```

Merge the properties of the initial face and others, with later faces taking priority.

This is used to combine the styles of multiple faces, and to resolve inheritance.

```
merge(a::NamedTuple, iterable)
```

Interpret an iterable of key-value pairs as a named tuple, and perform a merge.

```
julia> merge((a=1, b=2, c=3), [:b=>4, :d=>5])
(a = 1, b = 4, c = 3, d = 5)
```

source

```
merge(a::NamedTuple, bs::NamedTuple...)
```

Construct a new named tuple by merging two or more existing ones, in a left-associative manner. Merging proceeds left-to-right, between pairs of named tuples, and so the order of fields present in both the leftmost and rightmost named tuples take the same position as they are found in the leftmost named tuple. However, values are taken from matching fields in the rightmost named tuple that contains that field. Fields present in only the rightmost named tuple of a pair are appended at the end. A fallback is implemented for when only a single named tuple is supplied, with signature merge(a::NamedTuple).

Julia 1.1

Merging 3 or more NamedTuple requires at least Julia 1.1.

Examples

```
julia> merge((a=1, b=2, c=3), (b=4, d=5))
(a = 1, b = 4, c = 3, d = 5)
```

```
julia> merge((a=1, b=2), (b=3, c=(d=1,)), (c=(d=2,),))
(a = 1, b = 3, c = (d = 2,))
```

source

```
merge(d::AbstractDict, others::AbstractDict...)
```

Construct a merged collection from the given collections. If necessary, the types of the resulting collection will be promoted to accommodate the types of the merged collections. If the same key is present in another collection, the value for that key will be the value it has in the last collection listed. See also mergewith for custom handling of values with the same key.

Examples

```
julia> a = Dict("foo" => 0.0, "bar" => 42.0)
Dict{String, Float64} with 2 entries:
  "bar" => 42.0
  "foo" => 0.0

julia> b = Dict("baz" => 17, "bar" => 4711)
Dict{String, Int64} with 2 entries:
  "bar" => 4711
  "baz" => 17

julia> merge(a, b)
Dict{String, Float64} with 3 entries:
  "bar" => 4711.0
  "baz" => 17.0
  "foo" => 0.0

julia> merge(b, a)
Dict{String, Float64} with 3 entries:
  "bar" => 42.0
  "baz" => 17.0
  "foo" => 0.0
```

source

`Base.mergewith` – Function.

```
mergewith(combine, d::AbstractDict, others::AbstractDict...)
mergewith(combine)
```

Construct a merged collection from the given collections. If necessary, the types of the resulting collection will be promoted to accommodate the types of the merged collections. Values with the same key will be combined using the combiner function. The curried form `mergewith(combine)` returns the function `(args...) -> mergewith(combine, args...)`.

Julia 1.5

`mergewith` requires Julia 1.5 or later.

Examples

```
julia> a = Dict("foo" => 0.0, "bar" => 42.0)
Dict{String, Float64} with 2 entries:
  "bar" => 42.0
  "foo" => 0.0

julia> b = Dict("baz" => 17, "bar" => 4711)
Dict{String, Int64} with 2 entries:
  "bar" => 4711
  "baz" => 17

julia> mergewith(+, a, b)
Dict{String, Float64} with 3 entries:
  "bar" => 4753.0
```

```
  "baz" => 17.0
  "foo" => 0.0

julia> ans == mergewith(+)(a, b)
true

julia> mergewith(-, Dict(), Dict(:a=>1))  # Combining function only used if key is present in
↪  both
Dict{Any, Any} with 1 entry:
  :a => 1
```

source

Base.merge! – Function.

```
merge!(d::AbstractDict, others::AbstractDict...)
```

Update collection with pairs from the other collections. See also merge.

Examples

```
julia> d1 = Dict(1 => 2, 3 => 4);

julia> d2 = Dict(1 => 4, 4 => 5);

julia> merge!(d1, d2);

julia> d1
Dict{Int64, Int64} with 3 entries:
  4 => 5
  3 => 4
  1 => 4
```

source

Base.mergewith! – Function.

```
mergewith!(combine, d::AbstractDict, others::AbstractDict...) -> d
mergewith!(combine)
merge!(combine, d::AbstractDict, others::AbstractDict...) -> d
```

Update collection with pairs from the other collections. Values with the same key will be combined using the combiner function. The curried form `mergewith!(combine)` returns the function `(args...) -> mergewith!(combine, args...)`.

Method `merge!(combine::Union{Function,Type}, args...)` as an alias of `mergewith!(combine, args...)` is still available for backward compatibility.

Julia 1.5

`mergewith!` requires Julia 1.5 or later.

Examples

```
julia> d1 = Dict(1 => 2, 3 => 4);

julia> d2 = Dict(1 => 4, 4 => 5);

julia> mergewith!(+, d1, d2);

julia> d1
Dict{Int64, Int64} with 3 entries:
  4 => 5
  3 => 4
  1 => 6

julia> mergewith!(-, d1, d1);

julia> d1
Dict{Int64, Int64} with 3 entries:
  4 => 0
  3 => 0
  1 => 0

julia> foldl(mergewith!(+), [d1, d2]; init=Dict{Int64, Int64}())
Dict{Int64, Int64} with 3 entries:
  4 => 5
  3 => 0
  1 => 4
```

source

`Base.sizehint!` – Function.

```
sizehint!(s, n; first::Bool=false, shrink::Bool=true) -> s
```

Suggest that collection `s` reserve capacity for at least `n` elements. That is, if you expect that you're going to have to push a lot of values onto `s`, you can avoid the cost of incremental reallocation by doing it once up front; this can improve performance.

If `first` is `true`, then any additional space is reserved before the start of the collection. This way, subsequent calls to `pushfirst!` (instead of `push!`) may become faster. Supplying this keyword may result in an error if the collection is not ordered or if `pushfirst!` is not supported for this collection.

If `shrink=true` (the default), the collection's capacity may be reduced if its current capacity is greater than `n`.

See also `resize!`.

Notes on the performance model

For types that support `sizehint!`,

1. `push!` and `append!` methods generally may (but are not required to) preallocate extra storage. For types implemented in Base, they typically do, using a heuristic optimized for a general use case.
2. `sizehint!` may control this preallocation. Again, it typically does this for types in Base.
3. `empty!` is nearly costless (and O(1)) for types that support this kind of preallocation.

Julia 1.11

The `shrink` and `first` arguments were added in Julia 1.11.

source

Base.keytype – Function.

```
keytype(type)
```

Get the key type of a dictionary type. Behaves similarly to `eltype`.

Examples

```
julia> keytype(Dict(Int32(1) => "foo"))
Int32
```

source

```
keytype(T::Type{<:AbstractArray})
keytype(A::AbstractArray)
```

Return the key type of an array. This is equal to the `eltype` of the result of `keys(...)`, and is provided mainly for compatibility with the dictionary interface.

Examples

```
julia> keytype([1, 2, 3]) == Int
true

julia> keytype([1 2; 3 4])
CartesianIndex{2}
```

Julia 1.2

For arrays, this function requires at least Julia 1.2.

source

Base.valtype – Function.

```
valtype(type)
```

Get the value type of a dictionary type. Behaves similarly to `eltype`.

Examples

```
julia> valtype(Dict(Int32(1) => "foo"))
String
```

source

```
valtype(T::Type{<:AbstractArray})
valtype(A::AbstractArray)
```

Return the value type of an array. This is identical to `eltype` and is provided mainly for compatibility with the dictionary interface.

Examples

```
julia> valtype(["one", "two", "three"])
String
```

Julia 1.2

For arrays, this function requires at least Julia 1.2.

source

Fully implemented by:

- Dict
- IdDict
- WeakKeyDict

Partially implemented by:

- Set
- BitSet
- IdSet
- EnvDict
- Array
- BitArray
- ImmutableDict
- PersistentDict
- Iterators.Pairs

45.7 Set-Like Collections

Base.AbstractSet – Type.

```
AbstractSet{T}
```

Supertype for set-like types whose elements are of type T. Set, BitSet and other types are subtypes of this.

source

Base.Set – Type.

```
Set{T} <: AbstractSet{T}
```

Sets are mutable containers that provide fast membership testing.

Sets have efficient implementations of set operations such as `in`, `union` and `intersect`. Elements in a `Set` are unique, as determined by the elements' definition of `isequal`. The order of elements in a `Set` is an implementation detail and cannot be relied on.

See also: `AbstractSet`, `BitSet`, `Dict`, `push!`, `empty!`, `union!`, `in`, `isequal`

Examples

```
julia> s = Set("aaBca")
Set{Char} with 3 elements:
  'a'
  'c'
  'B'

julia> push!(s, 'b')
Set{Char} with 4 elements:
  'a'
  'b'
  'B'
  'c'

julia> s = Set([NaN, 0.0, 1.0, 2.0]);

julia> -0.0 in s # isequal(0.0, -0.0) is false
false

julia> NaN in s # isequal(NaN, NaN) is true
true
```

source

`Base.BitSet` – Type.

```
BitSet([itr])
```

Construct a sorted set of `Int`s generated by the given iterable object, or an empty set. Implemented as a bit string, and therefore designed for dense integer sets. If the set will be sparse (for example, holding a few very large integers), use `Set` instead.

source

`Base.IdSet` – Type.

```
IdSet{T}([itr])
IdSet()
```

`IdSet{T}()` constructs a set (see `Set`) using `===` as equality with values of type T.

In the example below, the values are all `isequal` so they get overwritten in the ordinary `Set`. The `IdSet` compares by `===` and so preserves the 3 different values.

Julia 1.11

Exported in Julia 1.11 and later.

Examples

```
julia> Set(Any[true, 1, 1.0])
Set{Any} with 1 element:
  1.0

julia> IdSet{Any}(Any[true, 1, 1.0])
IdSet{Any} with 3 elements:
  1.0
  1
  true
```

source

Base.union – Function.

```
union(s, itrs...)
∪(s, itrs...)
```

Construct an object containing all distinct elements from all of the arguments.

The first argument controls what kind of container is returned. If this is an array, it maintains the order in which elements first appear.

Unicode ∪ can be typed by writing \cup then pressing tab in the Julia REPL, and in many editors. This is an infix operator, allowing `s ∪ itr`.

See also unique, intersect, isdisjoint, vcat, Iterators.flatten.

Examples

```
julia> union([1, 2], [3])
3-element Vector{Int64}:
 1
 2
 3

julia> union([4 2 3 4 4], 1:3, 3.0)
4-element Vector{Float64}:
 4.0
 2.0
 3.0
 1.0

julia> (0, 0.0) ∪ (-0.0, NaN)
3-element Vector{Real}:
   0
  -0.0
 NaN

julia> union(Set([1, 2]), 2:3)
```

```
Set{Int64} with 3 elements:
  2
  3
  1
```

source

Base.union! – Function.

```
union!(s::Union{AbstractSet,AbstractVector}, itrs...)
```

Construct the union of passed in sets and overwrite s with the result. Maintain order with arrays.

Warning

Behavior can be unexpected when any mutated argument shares memory with any other argument.

Examples

```
julia> a = Set([3, 4, 5]);

julia> union!(a, 1:2:7);

julia> a
Set{Int64} with 5 elements:
  5
  4
  7
  3
  1
```

source

```
union!(s::IntDisjointSet{T}, x::T, y::T)
```

Merge the subset containing x and that containing y into one and return the root of the new set.

source

Base.intersect – Function.

```
intersect(s, itrs...)
∩(s, itrs...)
```

Construct the set containing those elements which appear in all of the arguments.

The first argument controls what kind of container is returned. If this is an array, it maintains the order in which elements first appear.

Unicode ∩ can be typed by writing \cap then pressing tab in the Julia REPL, and in many editors. This is an infix operator, allowing s ∩ itr.

See also setdiff, isdisjoint, issubset, issetequal.

Julia 1.8

As of Julia 1.8 intersect returns a result with the eltype of the type-promoted eltypes of the two inputs

Examples

```
julia> intersect([1, 2, 3], [3, 4, 5])
1-element Vector{Int64}:
 3

julia> intersect([1, 4, 4, 5, 6], [6, 4, 6, 7, 8])
2-element Vector{Int64}:
 4
 6

julia> intersect(1:16, 7:99)
7:16

julia> (0, 0.0) ∩ (-0.0, 0)
1-element Vector{Real}:
 0

julia> intersect(Set([1, 2]), BitSet([2, 3]), 1.0:10.0)
Set{Float64} with 1 element:
  2.0
```

source

Base.setdiff – Function.

```
setdiff(s, itrs...)
```

Construct the set of elements in `s` but not in any of the iterables in `itrs`. Maintain order with arrays. The result will have the same element type as `s`.

See also `setdiff!`, `union` and `intersect`.

Examples

```
julia> setdiff([1,2,3], [3,4,5])
2-element Vector{Int64}:
 1
 2

julia> setdiff([1,2,3], [1.0, 2.0])
1-element Vector{Int64}:
 3
```

source

Base.setdiff! – Function.

```
setdiff!(s, itrs...)
```

Remove from set s (in-place) each element of each iterable from `itrs`. Maintain order with arrays.

Warning

Behavior can be unexpected when any mutated argument shares memory with any other argument.

Examples

```
julia> a = Set([1, 3, 4, 5]);

julia> setdiff!(a, 1:2:6);

julia> a
Set{Int64} with 1 element:
  4
```

source

`Base.symdiff` – Function.

```
symdiff(s, itrs...)
```

Construct the symmetric difference of elements in the passed in sets. When s is not an `AbstractSet`, the order is maintained.

See also `symdiff!`, `setdiff`, `union` and `intersect`.

Examples

```
julia> symdiff([1,2,3], [3,4,5], [4,5,6])
3-element Vector{Int64}:
 1
 2
 6

julia> symdiff([1,2,1], [2, 1, 2])
Int64[]
```

source

`Base.symdiff!` – Function.

```
symdiff!(s::Union{AbstractSet,AbstractVector}, itrs...)
```

Construct the symmetric difference of the passed in sets, and overwrite s with the result. When s is an array, the order is maintained. Note that in this case the multiplicity of elements matters.

Warning

Behavior can be unexpected when any mutated argument shares memory with any other argument.

source

`Base.intersect!` – Function.

```
intersect!(s::Union{AbstractSet,AbstractVector}, itrs...)
```

Intersect all passed in sets and overwrite `s` with the result. Maintain order with arrays.

Warning

Behavior can be unexpected when any mutated argument shares memory with any other argument.

source

`Base.issubset` – Function.

```
issubset(a, b)::Bool
⊆(a, b)::Bool
⊇(b, a)::Bool
```

Determine whether every element of `a` is also in `b`, using `in`.

See also `⊊`, `⊈`, `∩`, `∪`, `contains`.

Examples

```
julia> issubset([1, 2], [1, 2, 3])
true

julia> [1, 2, 3] ⊆ [1, 2]
false

julia> [1, 2, 3] ⊇ [1, 2]
true
```

source

`Base.in!` – Function.

```
in!(x, s::AbstractSet)::Bool
```

If `x` is in `s`, return `true`. If not, push `x` into `s` and return `false`. This is equivalent to `in(x, s) ? true : (push!(s, x); false)`, but may have a more efficient implementation.

See also: `in`, `push!`, `Set`

Julia 1.11

This function requires at least 1.11.

Examples

```
julia> s = Set{Any}([1, 2, 3]); in!(4, s)
false

julia> length(s)
```

```
4

julia> in!(0x04, s)
true

julia> s
Set{Any} with 4 elements:
  4
  2
  3
  1
```

source

Base.:⊈ – Function.

```
⊈(a, b)::Bool
⊉(b, a)::Bool
```

Negation of ⊆ and ⊇, i.e. checks that a is not a subset of b.

See also issubset (⊆), ⊊.

Examples

```
julia> (1, 2) ⊈ (2, 3)
true

julia> (1, 2) ⊈ (1, 2, 3)
false
```

source

Base.:⊊ – Function.

```
⊊(a, b)::Bool
⊋(b, a)::Bool
```

Determines if a is a subset of, but not equal to, b.

See also issubset (⊆), ⊈.

Examples

```
julia> (1, 2) ⊊ (1, 2, 3)
true

julia> (1, 2) ⊊ (1, 2)
false
```

source

Base.issetequal – Function.

```
issetequal(x)
```

Create a function that compares its argument to x using `issetequal`, i.e. a function equivalent to `y -> issetequal(y, x)`. The returned function is of type `Base.Fix2{typeof(issetequal)}`, which can be used to implement specialized methods.

Julia 1.11

This functionality requires at least Julia 1.11.

source

```
issetequal(a, b)::Bool
```

Determine whether a and b have the same elements. Equivalent to `a ⊆ b && b ⊆ a` but more efficient when possible.

See also: `isdisjoint`, `union`.

Examples

```
julia> issetequal([1, 2], [1, 2, 3])
false

julia> issetequal([1, 2], [2, 1])
true
```

source

`Base.isdisjoint` – Function.

```
isdisjoint(x)
```

Create a function that compares its argument to x using `isdisjoint`, i.e. a function equivalent to `y -> isdisjoint(y, x)`. The returned function is of type `Base.Fix2{typeof(isdisjoint)}`, which can be used to implement specialized methods.

Julia 1.11

This functionality requires at least Julia 1.11.

source

```
isdisjoint(a, b)::Bool
```

Determine whether the collections a and b are disjoint. Equivalent to `isempty(a ∩ b)` but more efficient when possible.

See also: `intersect`, `isempty`, `issetequal`.

Julia 1.5

This function requires at least Julia 1.5.

Examples

```
julia> isdisjoint([1, 2], [2, 3, 4])
false

julia> isdisjoint([3, 1], [2, 4])
true
```

source

Fully implemented by:

- Set
- BitSet
- IdSet

Partially implemented by:

- Array

45.8 Deques

Base.push! – Function.

```
push!(collection, items...) -> collection
```

Insert one or more items in collection. If collection is an ordered container, the items are inserted at the end (in the given order).

Examples

```
julia> push!([1, 2, 3], 4, 5, 6)
6-element Vector{Int64}:
 1
 2
 3
 4
 5
 6
```

If collection is ordered, use append! to add all the elements of another collection to it. The result of the preceding example is equivalent to append!([1, 2, 3], [4, 5, 6]). For AbstractSet objects, union! can be used instead.

See sizehint! for notes about the performance model.

See also pushfirst!.

source

Base.pop! – Function.

```
pop!(collection, key[, default])
```

Delete and return the mapping for key if it exists in `collection`, otherwise return `default`, or throw an error if `default` is not specified.

Examples

```
julia> d = Dict("a"=>1, "b"=>2, "c"=>3);

julia> pop!(d, "a")
1

julia> pop!(d, "d")
ERROR: KeyError: key "d" not found
Stacktrace:
[...]

julia> pop!(d, "e", 4)
4
```

source

```
pop!(collection) -> item
```

Remove an item in `collection` and return it. If `collection` is an ordered container, the last item is returned; for unordered containers, an arbitrary element is returned.

See also: `popfirst!`, `popat!`, `delete!`, `deleteat!`, `splice!`, and `push!`.

Examples

```
julia> A=[1, 2, 3]
3-element Vector{Int64}:
 1
 2
 3

julia> pop!(A)
3

julia> A
2-element Vector{Int64}:
 1
 2

julia> S = Set([1, 2])
Set{Int64} with 2 elements:
  2
  1

julia> pop!(S)
2

julia> S
Set{Int64} with 1 element:
  1
```

```
julia> pop!(Dict(1=>2))
1 => 2
```

source

Base.popat! – Function.

```
popat!(a::Vector, i::Integer, [default])
```

Remove the item at the given `i` and return it. Subsequent items are shifted to fill the resulting gap. When `i` is not a valid index for `a`, return `default`, or throw an error if `default` is not specified.

See also: `pop!`, `popfirst!`, `deleteat!`, `splice!`.

Julia 1.5

This function is available as of Julia 1.5.

Examples

```
julia> a = [4, 3, 2, 1]; popat!(a, 2)
3

julia> a
3-element Vector{Int64}:
 4
 2
 1

julia> popat!(a, 4, missing)
missing

julia> popat!(a, 4)
ERROR: BoundsError: attempt to access 3-element Vector{Int64} at index [4]
[...]
```

source

Base.pushfirst! – Function.

```
pushfirst!(collection, items...) -> collection
```

Insert one or more `items` at the beginning of `collection`.

This function is called `unshift` in many other programming languages.

Examples

```
julia> pushfirst!([1, 2, 3, 4], 5, 6)
6-element Vector{Int64}:
 5
 6
 1
 2
 3
 4
```

source

Base.popfirst! – Function.

```
popfirst!(collection) -> item
```

Remove the first item from collection.

This function is called shift in many other programming languages.

See also: pop!, popat!, delete!.

Examples

```
julia> A = [1, 2, 3, 4, 5, 6]
6-element Vector{Int64}:
 1
 2
 3
 4
 5
 6

julia> popfirst!(A)
1

julia> A
5-element Vector{Int64}:
 2
 3
 4
 5
 6
```

source

Base.insert! – Function.

```
insert!(a::Vector, index::Integer, item)
```

Insert an item into a at the given index. index is the index of item in the resulting a.

See also: push!, replace, popat!, splice!.

Examples

```
julia> insert!(Any[1:6;], 3, "here")
7-element Vector{Any}:
 1
 2
  "here"
 3
 4
 5
 6
```

source

Base.deleteat! – Function.

```
deleteat!(a::Vector, inds)
```

Remove the items at the indices given by inds, and return the modified a. Subsequent items are shifted to fill the resulting gap.

inds can be either an iterator or a collection of sorted and unique integer indices, or a boolean vector of the same length as a with true indicating entries to delete.

Examples

```
julia> deleteat!([6, 5, 4, 3, 2, 1], 1:2:5)
3-element Vector{Int64}:
 5
 3
 1

julia> deleteat!([6, 5, 4, 3, 2, 1], [true, false, true, false, true, false])
3-element Vector{Int64}:
 5
 3
 1

julia> deleteat!([6, 5, 4, 3, 2, 1], (2, 2))
ERROR: ArgumentError: indices must be unique and sorted
Stacktrace:
[...]
```

source

```
deleteat!(a::Vector, i::Integer)
```

Remove the item at the given i and return the modified a. Subsequent items are shifted to fill the resulting gap.

See also: keepat!, delete!, popat!, splice!.

Examples

```
julia> deleteat!([6, 5, 4, 3, 2, 1], 2)
5-element Vector{Int64}:
 6
 4
 3
 2
 1
```

source

Base.keepat! – Function.

```
keepat!(a::Vector, m::AbstractVector{Bool})
keepat!(a::BitVector, m::AbstractVector{Bool})
```

The in-place version of logical indexing `a = a[m]`. That is, `keepat!(a, m)` on vectors of equal length `a` and `m` will remove all elements from `a` for which `m` at the corresponding index is `false`.

Examples

```
julia> a = [:a, :b, :c];

julia> keepat!(a, [true, false, true])
2-element Vector{Symbol}:
 :a
 :c

julia> a
2-element Vector{Symbol}:
 :a
 :c
```

source

```
keepat!(a::Vector, inds)
keepat!(a::BitVector, inds)
```

Remove the items at all the indices which are not given by `inds`, and return the modified `a`. Items which are kept are shifted to fill the resulting gaps.

Warning

Behavior can be unexpected when any mutated argument shares memory with any other argument.

`inds` must be an iterator of sorted and unique integer indices. See also `deleteat!`.

Julia 1.7

This function is available as of Julia 1.7.

Examples

```
julia> keepat!([6, 5, 4, 3, 2, 1], 1:2:5)
3-element Vector{Int64}:
 6
 4
 2
```

source

`Base.splice!` – Function.

```
splice!(a::Vector, indices, [replacement]) -> items
```

Remove items at specified indices, and return a collection containing the removed items. Subsequent items are shifted left to fill the resulting gaps. If specified, replacement values from an ordered collection will be spliced in place of the removed items; in this case, `indices` must be a `AbstractUnitRange`.

To insert `replacement` before an index n without removing any items, use `splice!(collection, n:n-1, replacement)`.

Warning

Behavior can be unexpected when any mutated argument shares memory with any other argument.

Julia 1.5

Prior to Julia 1.5, `indices` must always be a `UnitRange`.

Julia 1.8

Prior to Julia 1.8, `indices` must be a `UnitRange` if splicing in replacement values.

Examples

```
julia> A = [-1, -2, -3, 5, 4, 3, -1]; splice!(A, 4:3, 2)
Int64[]

julia> A
8-element Vector{Int64}:
 -1
 -2
 -3
  2
  5
  4
  3
 -1
```

source

```
splice!(a::Vector, index::Integer, [replacement]) -> item
```

Remove the item at the given index, and return the removed item. Subsequent items are shifted left to fill the resulting gap. If specified, replacement values from an ordered collection will be spliced in place of the removed item.

See also: `replace`, `delete!`, `deleteat!`, `pop!`, `popat!`.

Examples

```
julia> A = [6, 5, 4, 3, 2, 1]; splice!(A, 5)
2

julia> A
5-element Vector{Int64}:
```

```
 6
 5
 4
 3
 1

julia> splice!(A, 5, -1)
1

julia> A
5-element Vector{Int64}:
  6
  5
  4
  3
 -1

julia> splice!(A, 1, [-1, -2, -3])
6

julia> A
7-element Vector{Int64}:
 -1
 -2
 -3
  5
  4
  3
 -1
```

To insert `replacement` before an index `n` without removing any items, use `splice!(collection, n:n-1, replacement)`.

source

Base.`resize!` – Function.

```
resize!(a::Vector, n::Integer) -> a
```

Resize `a` to contain `n` elements. If `n` is smaller than the current collection length, the first `n` elements will be retained. If `n` is larger, the new elements are not guaranteed to be initialized.

Examples

```
julia> resize!([6, 5, 4, 3, 2, 1], 3)
3-element Vector{Int64}:
 6
 5
 4

julia> a = resize!([6, 5, 4, 3, 2, 1], 8);

julia> length(a)
8
```

```
julia> a[1:6]
6-element Vector{Int64}:
 6
 5
 4
 3
 2
 1
```

source

Base.append! – Function.

```
append!(collection, collections...) -> collection.
```

For an ordered container `collection`, add the elements of each `collections` to the end of it.

Julia 1.6

Specifying multiple collections to be appended requires at least Julia 1.6.

Examples

```
julia> append!([1], [2, 3])
3-element Vector{Int64}:
 1
 2
 3

julia> append!([1, 2, 3], [4, 5], [6])
6-element Vector{Int64}:
 1
 2
 3
 4
 5
 6
```

Use `push!` to add individual items to `collection` which are not already themselves in another collection. The result of the preceding example is equivalent to `push!([1, 2, 3], 4, 5, 6)`.

See `sizehint!` for notes about the performance model.

See also `vcat` for vectors, `union!` for sets, and `prepend!` and `pushfirst!` for the opposite order.

source

Base.prepend! – Function.

```
prepend!(a::Vector, collections...) -> collection
```

Insert the elements of each `collections` to the beginning of a.

When `collections` specifies multiple collections, order is maintained: elements of `collections[1]` will appear leftmost in a, and so on.

Julia 1.6

Specifying multiple collections to be prepended requires at least Julia 1.6.

Examples

```
julia> prepend!([3], [1, 2])
3-element Vector{Int64}:
 1
 2
 3

julia> prepend!([6], [1, 2], [3, 4, 5])
6-element Vector{Int64}:
 1
 2
 3
 4
 5
 6
```

source

Fully implemented by:

- `Vector` (a.k.a. 1-dimensional `Array`)
- `BitVector` (a.k.a. 1-dimensional `BitArray`)

45.9 Utility Collections

`Core.Pair` – Type.

```
Pair(x, y)
x => y
```

Construct a `Pair` object with type `Pair{typeof(x), typeof(y)}`. The elements are stored in the fields `first` and `second`. They can also be accessed via iteration (but a `Pair` is treated as a single "scalar" for broadcasting operations).

See also `Dict`.

Examples

```
julia> p = "foo" => 7
"foo" => 7

julia> typeof(p)
Pair{String, Int64}

julia> p.first
"foo"
```

```
julia> for x in p
           println(x)
       end
foo
7

julia> replace.(["xops", "oxps"], "x" => "o")
2-element Vector{String}:
 "oops"
 "oops"
```

source

Base.Pairs – Type.

```
Base.Pairs(values, keys) <: AbstractDict{eltype(keys), eltype(values)}
```

Transforms an indexable container into a Dictionary-view of the same data. Modifying the key-space of the underlying data may invalidate this object.

source

Chapter 46

Mathematics

46.1 Mathematical Operators

Base.:- – Method.

```
-(x)
```

Unary minus operator.

See also: abs, flipsign.

Examples

```
julia> -1
-1

julia> -(2)
-2

julia> -[1 2; 3 4]
2×2 Matrix{Int64}:
 -1  -2
 -3  -4

julia> -(true)  # promotes to Int
-1

julia> -(0x003)
0xfffd
```

source

Base.:+ – Function.

```
+(x, y...)
```

Addition operator.

Infix x+y+z+... calls this function with all arguments, i.e. +(x, y, z, ...), which by default then calls (x+y) + z + ... starting from the left.

Note that overflow is possible for most integer types, including the default `Int`, when adding large numbers.

Examples

```
julia> 1 + 20 + 4
25

julia> +(1, 20, 4)
25

julia> [1,2] + [3,4]
2-element Vector{Int64}:
 4
 6

julia> typemax(Int) + 1 < 0
true
```

source

```
dt::Date + t::Time -> DateTime
```

The addition of a `Date` with a `Time` produces a `DateTime`. The hour, minute, second, and millisecond parts of the `Time` are used along with the year, month, and day of the `Date` to create the new `DateTime`. Non-zero microseconds or nanoseconds in the `Time` type will result in an `InexactError` being thrown.

`Base.:-` – Method.

```
-(x, y)
```

Subtraction operator.

Examples

```
julia> 2 - 3
-1

julia> -(2, 4.5)
-2.5
```

source

`Base.:*` – Method.

```
*(x, y...)
```

Multiplication operator.

Infix `x*y*z*...` calls this function with all arguments, i.e. `*(x, y, z, ...)`, which by default then calls `(x*y) * z * ...` starting from the left.

Juxtaposition such as `2pi` also calls `*(2, pi)`. Note that this operation has higher precedence than a literal `*`. Note also that juxtaposition "0x..." (integer zero times a variable whose name starts with `x`) is forbidden as it clashes with unsigned integer literals: `0x01 isa UInt8`.

Note that overflow is possible for most integer types, including the default `Int`, when multiplying large numbers.

Examples

```
julia> 2 * 7 * 8
112

julia> *(2, 7, 8)
112

julia> [2 0; 0 3] * [1, 10]  # matrix * vector
2-element Vector{Int64}:
  2
 30

julia> 1/2pi, 1/2*pi  # juxtaposition has higher precedence
(0.15915494309189535, 1.5707963267948966)

julia> x = [1, 2]; x'x  # adjoint vector * vector
5
```

source

Base.:/ – Function.

```
/(x, y)
```

Right division operator: multiplication of x by the inverse of y on the right.

Gives floating-point results for integer arguments. See ÷ for integer division, or // for Rational results.

Examples

```
julia> 1/2
0.5

julia> 4/2
2.0

julia> 4.5/2
2.25
```

source

```
A / B
```

Matrix right-division: `A / B` is equivalent to `(B' \ A')'` where \ is the left-division operator. For square matrices, the result X is such that `A == X*B`.

See also: `rdiv!`.

Examples

```
julia> A = Float64[1 4 5; 3 9 2]; B = Float64[1 4 2; 3 4 2; 8 7 1];

julia> X = A / B
2×3 Matrix{Float64}:
 -0.65   3.75  -1.2
  3.25  -2.75   1.0

julia> isapprox(A, X*B)
true

julia> isapprox(X, A*pinv(B))
true
```

Base.:\ – Method.

```
\(x, y)
```

Left division operator: multiplication of y by the inverse of x on the left. Gives floating-point results for integer arguments.

Examples

```
julia> 3 \ 6
2.0

julia> inv(3) * 6
2.0

julia> A = [4 3; 2 1]; x = [5, 6];

julia> A \ x
2-element Vector{Float64}:
  6.5
 -7.0

julia> inv(A) * x
2-element Vector{Float64}:
  6.5
 -7.0
```

source

Base.:^ – Method.

```
^(x, y)
```

Exponentiation operator.

If x and y are integers, the result may overflow. To enter numbers in scientific notation, use Float64 literals such as 1.2e3 rather than 1.2 * 10^3.

If y is an Int literal (e.g. 2 in x^2 or -3 in x^-3), the Julia code x^y is transformed by the compiler to Base.literal_pow(^, x, Val(y)), to enable compile-time specialization on the value of the exponent. (As a default fallback we have Base.literal_pow(^, x, Val(y)) = ^(x,y), where usually ^

== Base.^ unless ^ has been defined in the calling namespace.) If y is a negative integer literal, then Base.literal_pow transforms the operation to inv(x)^-y by default, where -y is positive.

See also exp2, <<.

Examples

```
julia> 3^5
243

julia> 3^-1  # uses Base.literal_pow
0.3333333333333333

julia> p = -1;

julia> 3^p
ERROR: DomainError with -1:
Cannot raise an integer x to a negative power -1.
[...]

julia> 3.0^p
0.3333333333333333

julia> 10^19 > 0  # integer overflow
false

julia> big(10)^19 == 1e19
true
```

source

Base.fma – Function.

```
fma(x, y, z)
```

Computes x*y+z without rounding the intermediate result x*y. On some systems this is significantly more expensive than x*y+z. fma is used to improve accuracy in certain algorithms. See muladd.

source

Base.muladd – Function.

```
muladd(x, y, z)
```

Combined multiply-add: computes x*y+z, but allowing the add and multiply to be merged with each other or with surrounding operations for performance. For example, this may be implemented as an fma if the hardware supports it efficiently. The result can be different on different machines and can also be different on the same machine due to constant propagation or other optimizations. See fma.

Examples

```
julia> muladd(3, 2, 1)
7

julia> 3 * 2 + 1
7
```

source

```
muladd(A, y, z)
```

Combined multiply-add, `A*y .+ z`, for matrix-matrix or matrix-vector multiplication. The result is always the same size as `A*y`, but `z` may be smaller, or a scalar.

Julia 1.6

These methods require Julia 1.6 or later.

Examples

```
julia> A=[1.0 2.0; 3.0 4.0]; B=[1.0 1.0; 1.0 1.0]; z=[0, 100];

julia> muladd(A, B, z)
2×2 Matrix{Float64}:
   3.0    3.0
 107.0  107.0
```

`Base.inv` – Method.

```
inv(x)
```

Return the multiplicative inverse of `x`, such that `x*inv(x)` or `inv(x)*x` yields `one(x)` (the multiplicative identity) up to roundoff errors.

If `x` is a number, this is essentially the same as `one(x)/x`, but for some types `inv(x)` may be slightly more efficient.

Examples

```
julia> inv(2)
0.5

julia> inv(1 + 2im)
0.2 - 0.4im

julia> inv(1 + 2im) * (1 + 2im)
1.0 + 0.0im

julia> inv(2//3)
3//2
```

Julia 1.2

`inv(::Missing)` requires at least Julia 1.2.

source

`Base.div` – Function.

```
div(x, y)
÷(x, y)
```

The quotient from Euclidean (integer) division. Generally equivalent to a mathematical operation x/y without a fractional part.

See also: `cld`, `fld`, `rem`, `divrem`.

Examples

```
julia> 9 ÷ 4
2

julia> -5 ÷ 3
-1

julia> 5.0 ÷ 2
2.0

julia> div.(-5:5, 3)'
1×11 adjoint(::Vector{Int64}) with eltype Int64:
 -1  -1  -1  0  0  0  0  0  1  1  1
```

source

`Base.div` – Method.

```
div(x, y, r::RoundingMode=RoundToZero)
```

The quotient from Euclidean (integer) division. Computes `x / y`, rounded to an integer according to the rounding mode `r`. In other words, the quantity

```
round(x / y, r)
```

without any intermediate rounding.

Julia 1.4

The three-argument method taking a `RoundingMode` requires Julia 1.4 or later.

See also `fld` and `cld`, which are special cases of this function.

Julia 1.9

`RoundFromZero` requires at least Julia 1.9.

Examples:

```
julia> div(4, 3, RoundToZero) # Matches div(4, 3)
1
julia> div(4, 3, RoundDown) # Matches fld(4, 3)
1
julia> div(4, 3, RoundUp) # Matches cld(4, 3)
2
julia> div(5, 2, RoundNearest)
2
julia> div(5, 2, RoundNearestTiesAway)
```

```
3
julia> div(-5, 2, RoundNearest)
-2
julia> div(-5, 2, RoundNearestTiesAway)
-3
julia> div(-5, 2, RoundNearestTiesUp)
-2
julia> div(4, 3, RoundFromZero)
2
julia> div(-4, 3, RoundFromZero)
-2
```

Because `div(x, y)` implements strictly correct truncated rounding based on the true value of floating-point numbers, unintuitive situations can arise. For example:

```
julia> div(6.0, 0.1)
59.0
julia> 6.0 / 0.1
60.0
julia> 6.0 / big(0.1)
59.99999999999999666933092612453056361837965690217069245739573412231113406246995
```

What is happening here is that the true value of the floating-point number written as `0.1` is slightly larger than the numerical value 1/10 while `6.0` represents the number 6 precisely. Therefore the true value of `6.0 / 0.1` is slightly less than 60. When doing division, this is rounded to precisely `60.0`, but `div(6.0, 0.1, RoundToZero)` always truncates the true value, so the result is `59.0`.

source

`Base.fld` – Function.

```
fld(x, y)
```

Largest integer less than or equal to `x / y`. Equivalent to `div(x, y, RoundDown)`.

See also `div`, `cld`, `fld1`.

Examples

```
julia> fld(7.3, 5.5)
1.0

julia> fld.(-5:5, 3)'
1×11 adjoint(::Vector{Int64}) with eltype Int64:
 -2  -2  -1  -1  -1  0  0  0  1  1  1
```

Because `fld(x, y)` implements strictly correct floored rounding based on the true value of floating-point numbers, unintuitive situations can arise. For example:

```
julia> fld(6.0, 0.1)
59.0
julia> 6.0 / 0.1
60.0
julia> 6.0 / big(0.1)
59.99999999999999666933092612453056361837965690217069245739573412231113406246995
```

What is happening here is that the true value of the floating-point number written as `0.1` is slightly larger than the numerical value 1/10 while `6.0` represents the number 6 precisely. Therefore the true value of `6.0 / 0.1` is slightly less than 60. When doing division, this is rounded to precisely `60.0`, but `fld(6.0, 0.1)` always takes the floor of the true value, so the result is `59.0`.

source

`Base.cld` – Function.

```
cld(x, y)
```

Smallest integer larger than or equal to `x / y`. Equivalent to `div(x, y, RoundUp)`.

See also `div`, `fld`.

Examples

```
julia> cld(5.5, 2.2)
3.0

julia> cld.(-5:5, 3)'
1×11 adjoint(::Vector{Int64}) with eltype Int64:
 -1  -1  -1  0  0  0  1  1  1  2  2
```

source

`Base.mod` – Function.

```
rem(x::Integer, T::Type{<:Integer})::T
mod(x::Integer, T::Type{<:Integer})::T
%(x::Integer, T::Type{<:Integer})::T
```

Find `y::T` such that x ≡ y (mod n), where n is the number of integers representable in T, and y is an integer in `[typemin(T),typemax(T)]`. If T can represent any integer (e.g. `T == BigInt`), then this operation corresponds to a conversion to T.

Examples

```
julia> x = 129 % Int8
-127

julia> typeof(x)
Int8

julia> x = 129 % BigInt
129

julia> typeof(x)
BigInt
```

source

```
mod(x, y)
rem(x, y, RoundDown)
```

The reduction of x modulo y, or equivalently, the remainder of x after floored division by y, i.e. `x - y*fld(x,y)` if computed without intermediate rounding.

The result will have the same sign as y if `isfinite(y)`, and magnitude less than `abs(y)` (with some exceptions, see note below).

Note

When used with floating point values, the exact result may not be representable by the type, and so rounding error may occur. In particular, if the exact result is very close to y, then it may be rounded to y.

See also: `rem`, `div`, `fld`, `mod1`, `invmod`.

```
julia> mod(8, 3)
2

julia> mod(9, 3)
0

julia> mod(8.9, 3)
2.9000000000000004

julia> mod(eps(), 3)
2.220446049250313e-16

julia> mod(-eps(), 3)
3.0

julia> mod.(-5:5, 3)'
1×11 adjoint(::Vector{Int64}) with eltype Int64:
 1  2  0  1  2  0  1  2  0  1  2
```

source

```
mod(x::Integer, r::AbstractUnitRange)
```

Find y in the range r such that x ≡ y (mod n), where `n = length(r)`, i.e. `y = mod(x - first(r), n) + first(r)`.

See also `mod1`.

Examples

```
julia> mod(0, Base.OneTo(3))  # mod1(0, 3)
3

julia> mod(3, 0:2)  # mod(3, 3)
0
```

Julia 1.3

This method requires at least Julia 1.3.

source

Base.rem – Function.

```
rem(x, y)
%(x, y)
```

Remainder from Euclidean division, returning a value of the same sign as x, and smaller in magnitude than y. This value is always exact.

See also: div, mod, mod1, divrem.

Examples

```
julia> x = 15; y = 4;

julia> x % y
3

julia> x == div(x, y) * y + rem(x, y)
true

julia> rem.(-5:5, 3)'
1×11 adjoint(::Vector{Int64}) with eltype Int64:
 -2  -1  0  -2  -1  0  1  2  0  1  2
```

source

Base.rem – Method.

```
rem(x, y, r::RoundingMode=RoundToZero)
```

Compute the remainder of x after integer division by y, with the quotient rounded according to the rounding mode r. In other words, the quantity

```
x - y * round(x / y, r)
```

without any intermediate rounding.

- if r == RoundNearest, then the result is exact, and in the interval $[-|y|/2, |y|/2]$. See also RoundNearest.
- if r == RoundToZero (default), then the result is exact, and in the interval $[0, |y|)$ if x is positive, or $(-|y|, 0]$ otherwise. See also RoundToZero.
- if r == RoundDown, then the result is in the interval $[0, y)$ if y is positive, or $(y, 0]$ otherwise. The result may not be exact if x and y have different signs, and abs(x) < abs(y). See also RoundDown.
- if r == RoundUp, then the result is in the interval $(-y, 0]$ if y is positive, or $[0, -y)$ otherwise. The result may not be exact if x and y have the same sign, and abs(x) < abs(y). See also RoundUp.
- if r == RoundFromZero, then the result is in the interval $(-y, 0]$ if y is positive, or $[0, -y)$ otherwise. The result may not be exact if x and y have the same sign, and abs(x) < abs(y). See also RoundFromZero.

Julia 1.9

RoundFromZero requires at least Julia 1.9.

Examples:

```
julia> x = 9; y = 4;

julia> x % y  # same as rem(x, y)
1

julia> x ÷ y  # same as div(x, y)
2

julia> x == div(x, y) * y + rem(x, y)
true
```

source

Base.Math.rem2pi – Function.

```
rem2pi(x, r::RoundingMode)
```

Compute the remainder of x after integer division by 2π, with the quotient rounded according to the rounding mode r. In other words, the quantity

```
x - 2π*round(x/(2π),r)
```

without any intermediate rounding. This internally uses a high precision approximation of 2π, and so will give a more accurate result than rem(x,2π,r)

- if r == RoundNearest, then the result is in the interval $[-,]$. This will generally be the most accurate result. See also RoundNearest.
- if r == RoundToZero, then the result is in the interval $[0, 2]$ if x is positive,. or $[-2, 0]$ otherwise. See also RoundToZero.
- if r == RoundDown, then the result is in the interval $[0, 2]$. See also RoundDown.
- if r == RoundUp, then the result is in the interval $[-2, 0]$. See also RoundUp.

Examples

```
julia> rem2pi(7pi/4, RoundNearest)
-0.7853981633974485

julia> rem2pi(7pi/4, RoundDown)
5.497787143782138
```

source

Base.Math.mod2pi – Function.

```
mod2pi(x)
```

Modulus after division by 2π, returning in the range $[0, 2)$.

This function computes a floating point representation of the modulus after division by numerically exact 2π, and is therefore not exactly the same as `mod(x,2π)`, which would compute the modulus of `x` relative to division by the floating-point number 2π.

Note

Depending on the format of the input value, the closest representable value to 2π may be less than 2π. For example, the expression `mod2pi(2π)` will not return `0`, because the intermediate value of `2*π` is a `Float64` and `2*Float64(π) < 2*big(π)`. See `rem2pi` for more refined control of this behavior.

Examples

```
julia> mod2pi(9*pi/4)
0.7853981633974481
```

source

Base.divrem – Function.

```
divrem(x, y, r::RoundingMode=RoundToZero)
```

The quotient and remainder from Euclidean division. Equivalent to `(div(x, y, r), rem(x, y, r))`. Equivalently, with the default value of `r`, this call is equivalent to `(x ÷ y, x % y)`.

See also: `fldmod`, `cld`.

Examples

```
julia> divrem(3, 7)
(0, 3)

julia> divrem(7, 3)
(2, 1)
```

source

Base.fldmod – Function.

```
fldmod(x, y)
```

The floored quotient and modulus after division. A convenience wrapper for `divrem(x, y, RoundDown)`. Equivalent to `(fld(x, y), mod(x, y))`.

See also: `fld`, `cld`, `fldmod1`.

source

Base.fld1 – Function.

```
fld1(x, y)
```

Flooring division, returning a value consistent with `mod1(x,y)`

See also `mod1`, `fldmod1`.

Examples

```
julia> x = 15; y = 4;

julia> fld1(x, y)
4

julia> x == fld(x, y) * y + mod(x, y)
true

julia> x == (fld1(x, y) - 1) * y + mod1(x, y)
true
```

source

`Base.mod1` – Function.

```
mod1(x, y)
```

Modulus after flooring division, returning a value `r` such that `mod(r, y) == mod(x, y)` in the range $(0, y]$ for positive `y` and in the range $[y, 0)$ for negative `y`.

With integer arguments and positive `y`, this is equal to `mod(x, 1:y)`, and hence natural for 1-based indexing. By comparison, `mod(x, y) == mod(x, 0:y-1)` is natural for computations with offsets or strides.

See also `mod`, `fld1`, `fldmod1`.

Examples

```
julia> mod1(4, 2)
2

julia> mod1.(-5:5, 3)'
1×11 adjoint(::Vector{Int64}) with eltype Int64:
 1  2  3  1  2  3  1  2  3  1  2

julia> mod1.([-0.1, 0, 0.1, 1, 2, 2.9, 3, 3.1]', 3)
1×8 Matrix{Float64}:
 2.9  3.0  0.1  1.0  2.0  2.9  3.0  0.1
```

source

`Base.fldmod1` – Function.

```
fldmod1(x, y)
```

Return `(fld1(x,y), mod1(x,y))`.

See also `fld1`, `mod1`.

source

Base.://– Function.

```
//(num, den)
```

Divide two integers or rational numbers, giving a Rational result. More generally, // can be used for exact rational division of other numeric types with integer or rational components, such as complex numbers with integer components.

Note that floating-point (AbstractFloat) arguments are not permitted by // (even if the values are rational). The arguments must be subtypes of Integer, Rational, or composites thereof.

Examples

```
julia> 3 // 5
3//5

julia> (3 // 5) // (2 // 1)
3//10

julia> (1+2im) // (3+4im)
11//25 + 2//25*im

julia> 1.0 // 2
ERROR: MethodError: no method matching //(::Float64, ::Int64)
[...]
```

source

Base.rationalize – Function.

```
rationalize([T<:Integer=Int,] x; tol::Real=eps(x))
```

Approximate floating point number x as a Rational number with components of the given integer type. The result will differ from x by no more than tol.

Examples

```
julia> rationalize(5.6)
28//5

julia> a = rationalize(BigInt, 10.3)
103//10

julia> typeof(numerator(a))
BigInt
```

source

Base.numerator – Function.

```
numerator(x)
```

Numerator of the rational representation of x.

Examples

```
julia> numerator(2//3)
2

julia> numerator(4)
4
```

source

Base.denominator – Function.

```
denominator(x)
```

Denominator of the rational representation of x.

Examples

```
julia> denominator(2//3)
3

julia> denominator(4)
1
```

source

Base.:<< – Function.

```
<<(B::BitVector, n)::BitVector
```

Left bit shift operator, `B << n`. For `n >= 0`, the result is B with elements shifted n positions backwards, filling with `false` values. If `n < 0`, elements are shifted forwards. Equivalent to `B >> -n`.

Examples

```
julia> B = BitVector([true, false, true, false, false])
5-element BitVector:
 1
 0
 1
 0
 0

julia> B << 1
5-element BitVector:
 0
 1
 0
 0
 0

julia> B << -1
5-element BitVector:
 0
 1
 0
 1
 0
```

source

```
<<(x, n)
```

Left bit shift operator, `x << n`. For `n >= 0`, the result is `x` shifted left by `n` bits, filling with `0`s. This is equivalent to `x * 2^n`. For `n < 0`, this is equivalent to `x >> -n`.

Examples

```
julia> Int8(3) << 2
12

julia> bitstring(Int8(3))
"00000011"

julia> bitstring(Int8(12))
"00001100"
```

See also `>>`, `>>>`, `exp2`, `ldexp`.

source

Base.:>> – Function.

```
>>(B::BitVector, n)::BitVector
```

Right bit shift operator, `B >> n`. For `n >= 0`, the result is `B` with elements shifted `n` positions forward, filling with `false` values. If `n < 0`, elements are shifted backwards. Equivalent to `B << -n`.

Examples

```
julia> B = BitVector([true, false, true, false, false])
5-element BitVector:
 1
 0
 1
 0
 0

julia> B >> 1
5-element BitVector:
 0
 1
 0
 1
 0

julia> B >> -1
5-element BitVector:
 0
 1
 0
 0
 0
```

source

```
>>(x, n)
```

Right bit shift operator, `x >> n`. For `n >= 0`, the result is x shifted right by n bits, filling with `0`s if `x >= 0`, 1s if `x < 0`, preserving the sign of x. This is equivalent to `fld(x, 2^n)`. For `n < 0`, this is equivalent to `x << -n`.

Examples

```
julia> Int8(13) >> 2
3

julia> bitstring(Int8(13))
"00001101"

julia> bitstring(Int8(3))
"00000011"

julia> Int8(-14) >> 2
-4

julia> bitstring(Int8(-14))
"11110010"

julia> bitstring(Int8(-4))
"11111100"
```

See also >>>, <<.

source

Base.:>>> – Function.

```
>>>(B::BitVector, n)::BitVector
```

Unsigned right bitshift operator, `B >>> n`. Equivalent to `B >> n`. See >> for details and examples.

source

```
>>>(x, n)
```

Unsigned right bit shift operator, `x >>> n`. For `n >= 0`, the result is x shifted right by n bits, filling with `0`s. For `n < 0`, this is equivalent to `x << -n`.

For Unsigned integer types, this is equivalent to >>. For Signed integer types, this is equivalent to `signed(unsigned(x) >> n)`.

Examples

```
julia> Int8(-14) >>> 2
60

julia> bitstring(Int8(-14))
"11110010"
```

```
julia> bitstring(Int8(60))
"00111100"
```

BigInts are treated as if having infinite size, so no filling is required and this is equivalent to >>.

See also >>, <<.

source

Base.bitrotate – Function.

```
bitrotate(x::Base.BitInteger, k::Integer)
```

bitrotate(x, k) implements bitwise rotation. It returns the value of x with its bits rotated left k times. A negative value of k will rotate to the right instead.

Julia 1.5

This function requires Julia 1.5 or later.

See also: <<, circshift, BitArray.

```
julia> bitrotate(UInt8(114), 2)
0xc9

julia> bitstring(bitrotate(0b01110010, 2))
"11001001"

julia> bitstring(bitrotate(0b01110010, -2))
"10011100"

julia> bitstring(bitrotate(0b01110010, 8))
"01110010"
```

source

Base.:(:) – Method.

```
(:)(start, [step], stop)
```

Range operator. a:b constructs a range from a to b with a step size equal to +1, which produces:

- a UnitRange when a and b are integers, or
- a StepRange when a and b are characters, or
- a StepRangeLen when a and/or b are floating-point.

a:s:b is similar but uses a step size of s (a StepRange or StepRangeLen). See also range for more control.

To create a descending range, use reverse(a:b) or a negative step size, e.g. b:-1:a. Otherwise, when b < a, an empty range will be constructed and normalized to a:a-1.

The operator : is also used in indexing to select whole dimensions, e.g. in A[:, 1].

`:` is also used to `quote` code, e.g. `:(x + y) isa Expr` and `:x isa Symbol`. Since `:2 isa Int`, it does *not* create a range in indexing: `v[:2] == v[2] != v[begin:2]`.

source

Base.:(:) – Method.

```
(:)(start::CartesianIndex, [step::CartesianIndex], stop::CartesianIndex)
```

Construct `CartesianIndices` from two `CartesianIndex` and an optional step.

Julia 1.1

This method requires at least Julia 1.1.

Julia 1.6

The step range method start:step:stop requires at least Julia 1.6.

Examples

```
julia> I = CartesianIndex(2,1);

julia> J = CartesianIndex(3,3);

julia> I:J
CartesianIndices((2:3, 1:3))

julia> I:CartesianIndex(1, 2):J
CartesianIndices((2:1:3, 1:2:3))
```

source

Base.range – Function.

```
range(start, stop, length)
range(start, stop; length, step)
range(start; length, stop, step)
range(;start, length, stop, step)
```

Construct a specialized array with evenly spaced elements and optimized storage (an `AbstractRange`) from the arguments. Mathematically a range is uniquely determined by any three of `start`, `step`, `stop` and `length`. Valid invocations of range are:

- Call `range` with any three of `start`, `step`, `stop`, `length`.
- Call `range` with two of `start`, `stop`, `length`. In this case `step` will be assumed to be positive one. If both arguments are Integers, a `UnitRange` will be returned.
- Call `range` with one of `stop` or `length`. `start` and `step` will be assumed to be positive one.

To construct a descending range, specify a negative step size, e.g. `range(5, 1; step = -1) => [5,4,3,2,1]`. Otherwise, a `stop` value less than the `start` value, with the default `step` of +1, constructs

an empty range. Empty ranges are normalized such that the `stop` is one less than the `start`, e.g. `range(5, 1) == 5:4`.

See Extended Help for additional details on the returned type. See also `logrange` for logarithmically spaced points.

Examples

```
julia> range(1, length=100)
1:100

julia> range(1, stop=100)
1:100

julia> range(1, step=5, length=100)
1:5:496

julia> range(1, step=5, stop=100)
1:5:96

julia> range(1, 10, length=101)
1.0:0.09:10.0

julia> range(1, 100, step=5)
1:5:96

julia> range(stop=10, length=5)
6:10

julia> range(stop=10, step=1, length=5)
6:1:10

julia> range(start=1, step=1, stop=10)
1:1:10

julia> range(; length = 10)
Base.OneTo(10)

julia> range(; stop = 6)
Base.OneTo(6)

julia> range(; stop = 6.5)
1.0:1.0:6.0
```

If `length` is not specified and `stop - start` is not an integer multiple of `step`, a range that ends before `stop` will be produced.

```
julia> range(1, 3.5, step=2)
1.0:2.0:3.0
```

Special care is taken to ensure intermediate values are computed rationally. To avoid this induced overhead, see the `LinRange` constructor.

Julia 1.1

stop as a positional argument requires at least Julia 1.1.

Julia 1.7

The versions without keyword arguments and start as a keyword argument require at least Julia 1.7.

Julia 1.8

The versions with stop as a sole keyword argument, or length as a sole keyword argument require at least Julia 1.8.

Extended Help

range will produce a Base.OneTo when the arguments are Integers and

- Only length is provided
- Only stop is provided

range will produce a UnitRange when the arguments are Integers and

- Only start and stop are provided
- Only length and stop are provided

A UnitRange is not produced if step is provided even if specified as one.

source

Base.OneTo – Type.

```
Base.OneTo(n)
```

Define an AbstractUnitRange that behaves like 1:n, with the added distinction that the lower limit is guaranteed (by the type system) to be 1.

source

Base.StepRangeLen – Type.

```
StepRangeLen(         ref::R, step::S, len, [offset=1]) where {  R,S}
StepRangeLen{T,R,S}(  ref::R, step::S, len, [offset=1]) where {T,R,S}
StepRangeLen{T,R,S,L}(ref::R, step::S, len, [offset=1]) where {T,R,S,L}
```

A range r where r[i] produces values of type T (in the first form, T is deduced automatically), parameterized by a reference value, a step, and the length. By default ref is the starting value r[1], but alternatively you can supply it as the value of r[offset] for some other index 1 <= offset <= len. The syntax a:b or a:b:c, where any of a, b, or c are floating-point numbers, creates a StepRangeLen.

Julia 1.7

The 4th type parameter L requires at least Julia 1.7.

source

Base.logrange – Function.

```
logrange(start, stop, length)
logrange(start, stop; length)
```

Construct a specialized array whose elements are spaced logarithmically between the given endpoints. That is, the ratio of successive elements is a constant, calculated from the length.

This is similar to geomspace in Python. Unlike PowerRange in Mathematica, you specify the number of elements not the ratio. Unlike logspace in Python and Matlab, the start and stop arguments are always the first and last elements of the result, not powers applied to some base.

Examples

```
julia> logrange(10, 4000, length=3)
3-element Base.LogRange{Float64, Base.TwicePrecision{Float64}}:
 10.0, 200.0, 4000.0

julia> ans[2] ≈ sqrt(10 * 4000)  # middle element is the geometric mean
true

julia> range(10, 40, length=3)[2] ≈ (10 + 40)/2  # arithmetic mean
true

julia> logrange(1f0, 32f0, 11)
11-element Base.LogRange{Float32, Float64}:
 1.0, 1.41421, 2.0, 2.82843, 4.0, 5.65685, 8.0, 11.3137, 16.0, 22.6274, 32.0

julia> logrange(1, 1000, length=4) ≈ 10 .^ (0:3)
true
```

See the LogRange type for further details.

See also range for linearly spaced points.

Julia 1.11

This function requires at least Julia 1.11.

source

Base.LogRange – Type.

```
LogRange{T}(start, stop, len) <: AbstractVector{T}
```

A range whose elements are spaced logarithmically between start and stop, with spacing controlled by len. Returned by logrange.

Like LinRange, the first and last elements will be exactly those provided, but intermediate values may have small floating-point errors. These are calculated using the logs of the endpoints, which are stored on construction, often in higher precision than T.

Examples

```
julia> logrange(1, 4, length=5)
5-element Base.LogRange{Float64, Base.TwicePrecision{Float64}}:
 1.0, 1.41421, 2.0, 2.82843, 4.0

julia> Base.LogRange{Float16}(1, 4, 5)
5-element Base.LogRange{Float16, Float64}:
 1.0, 1.414, 2.0, 2.828, 4.0

julia> logrange(1e-310, 1e-300, 11)[1:2:end]
6-element Vector{Float64}:
 1.0e-310
 9.999999999999974e-309
 9.999999999999981e-307
 9.999999999999988e-305
 9.999999999999994e-303
 1.0e-300

julia> prevfloat(1e-308, 5) == ans[2]
true
```

Note that integer eltype T is not allowed. Use for instance `round.(Int, xs)`, or explicit powers of some integer base:

```
julia> xs = logrange(1, 512, 4)
4-element Base.LogRange{Float64, Base.TwicePrecision{Float64}}:
 1.0, 8.0, 64.0, 512.0

julia> 2 .^ (0:3:9) |> println
[1, 8, 64, 512]
```

Julia 1.11

This type requires at least Julia 1.11.

source

`Base.:(==)` – Function.

```
==(x, y)
```

Generic equality operator. Falls back to `===`. Should be implemented for all types with a notion of equality, based on the abstract value that an instance represents. For example, all numeric types are compared by numeric value, ignoring type. Strings are compared as sequences of characters, ignoring encoding. Collections of the same type generally compare their key sets, and if those are ==, then compare the values for each of those keys, returning true if all such pairs are ==. Other properties are typically not taken into account (such as the exact type).

This operator follows IEEE semantics for floating-point numbers: `0.0 == -0.0` and `NaN != NaN`.

The result is of type `Bool`, except when one of the operands is `missing`, in which case `missing` is returned (three-valued logic). Collections generally implement three-valued logic akin to `all`, returning missing if any operands contain missing values and all other pairs are equal. Use `isequal` or `===` to always get a `Bool` result.

Implementation

New numeric types should implement this function for two arguments of the new type, and handle comparison to other types via promotion rules where possible.

Equality and hashing are intimately related; two values that are considered `isequal` **must** have the same `hash` and by default `isequal` falls back to `==`. If a type customizes the behavior of `==` and/or `isequal`, then `hash` must be similarly implemented to ensure `isequal` and `hash` agree. `Sets`, `Dicts`, and many other internal implementations assume that this invariant holds.

If some type defines `==`, `isequal`, and `isless` then it should also implement `<` to ensure consistency of comparisons.

source

`Base.:!=` – Function.

```
!=(x)
```

Create a function that compares its argument to x using `!=`, i.e. a function equivalent to `y -> y != x`. The returned function is of type `Base.Fix2{typeof(!=)}`, which can be used to implement specialized methods.

Julia 1.2

This functionality requires at least Julia 1.2.

source

```
!=(x, y)
≠(x,y)
```

Not-equals comparison operator. Always gives the opposite answer as `==`.

Implementation

New types should generally not implement this, and rely on the fallback definition `!=(x,y) = !(x==y)` instead.

Examples

```
julia> 3 != 2
true

julia> "foo" ≠ "foo"
false
```

source

`Core.:!==` – Function.

```
!==(x, y)
≢(x,y)
```

Always gives the opposite answer as `===`.

Examples

```
julia> a = [1, 2]; b = [1, 2];

julia> a ≢ b
true

julia> a ≢ a
false
```

source

Base.:< – Function.

```
<(x)
```

Create a function that compares its argument to x using <, i.e. a function equivalent to y -> y < x. The returned function is of type Base.Fix2{typeof(<)}, which can be used to implement specialized methods.

Julia 1.2

This functionality requires at least Julia 1.2.

source

```
<(x, y)
```

Less-than comparison operator. Falls back to isless. Because of the behavior of floating-point NaN values, this operator implements a partial order.

Implementation

New types with a canonical partial order should implement this function for two arguments of the new type. Types with a canonical total order should implement isless instead.

See also isunordered.

Examples

```
julia> 'a' < 'b'
true

julia> "abc" < "abd"
true

julia> 5 < 3
false
```

source

Base.:<= – Function.

```
<=(x)
```

Create a function that compares its argument to x using <=, i.e. a function equivalent to y -> y <= x. The returned function is of type Base.Fix2{typeof(<=)}, which can be used to implement specialized methods.

Julia 1.2

This functionality requires at least Julia 1.2.

source

```
<=(x, y)
≤(x,y)
```

Less-than-or-equals comparison operator. Falls back to (x < y) | (x == y).

Examples

```
julia> 'a' <= 'b'
true

julia> 7 ≤ 7 ≤ 9
true

julia> "abc" ≤ "abc"
true

julia> 5 <= 3
false
```

source

Base.:> – Function.

```
>(x)
```

Create a function that compares its argument to x using >, i.e. a function equivalent to y -> y > x. The returned function is of type Base.Fix2{typeof(>)}, which can be used to implement specialized methods.

Julia 1.2

This functionality requires at least Julia 1.2.

source

```
>(x, y)
```

Greater-than comparison operator. Falls back to y < x.

Implementation

Generally, new types should implement < instead of this function, and rely on the fallback definition >(x, y) = y < x.

Examples

```
julia> 'a' > 'b'
false

julia> 7 > 3 > 1
true

julia> "abc" > "abd"
false

julia> 5 > 3
true
```

source

Base.:>= – Function.

```
>=(x)
```

Create a function that compares its argument to x using >=, i.e. a function equivalent to `y -> y >= x`. The returned function is of type `Base.Fix2{typeof(>=)}`, which can be used to implement specialized methods.

Julia 1.2

This functionality requires at least Julia 1.2.

source

```
>=(x, y)
≥(x,y)
```

Greater-than-or-equals comparison operator. Falls back to `y <= x`.

Implementation

New types should prefer to implement <= instead of this function, and rely on the fallback definition `>=(x, y) = y <= x`.

Furthermore, in many cases it is enough to implement just < and ==, relying on the fallback definitions of both <= and >=.

Examples

```
julia> 'a' >= 'b'
false

julia> 7 ≥ 7 ≥ 3
true

julia> "abc" ≥ "abc"
true

julia> 5 >= 3
true
```

source

Base.cmp – Function.

```
cmp(a::AbstractString, b::AbstractString)::Int
```

Compare two strings. Return 0 if both strings have the same length and the character at each index is the same in both strings. Return -1 if a is a prefix of b, or if a comes before b in alphabetical order. Return 1 if b is a prefix of a, or if b comes before a in alphabetical order (technically, lexicographical order by Unicode code points).

Examples

```
julia> cmp("abc", "abc")
0

julia> cmp("ab", "abc")
-1

julia> cmp("abc", "ab")
1

julia> cmp("ab", "ac")
-1

julia> cmp("ac", "ab")
1

julia> cmp("α", "a")
1

julia> cmp("b", "β")
-1
```

source

```
cmp(<, x, y)
```

Return -1, 0, or 1 depending on whether x is less than, equal to, or greater than y, respectively. The first argument specifies a less-than comparison function to use.

source

```
cmp(x,y)
```

Return -1, 0, or 1 depending on whether x is less than, equal to, or greater than y, respectively. Uses the total order implemented by `isless`.

Examples

```
julia> cmp(1, 2)
-1

julia> cmp(2, 1)
```

```
1

julia> cmp(2+im, 3-im)
ERROR: MethodError: no method matching isless(::Complex{Int64}, ::Complex{Int64})
[...]
```

source

Base.:~ – Function.

```
~(x)
```

Bitwise not.

See also: !, &, |.

Examples

```
julia> ~4
-5

julia> ~10
-11

julia> ~true
false
```

source

Base.:& – Function.

```
x & y
```

Bitwise and. Implements three-valued logic, returning missing if one operand is missing and the other is true. Add parentheses for function application form: (&)(x, y).

See also: |, xor, &&.

Examples

```
julia> 4 & 10
0

julia> 4 & 12
4

julia> true & missing
missing

julia> false & missing
false
```

source

Base.:| – Function.

```
x | y
```

Bitwise or. Implements three-valued logic, returning `missing` if one operand is `missing` and the other is `false`.

See also: `&`, `xor`, `||`.

Examples

```
julia> 4 | 10
14

julia> 4 | 1
5

julia> true | missing
true

julia> false | missing
missing
```

source

`Base.xor` – Function.

```
xor(x, y)
⊻(x, y)
```

Bitwise exclusive or of `x` and `y`. Implements three-valued logic, returning `missing` if one of the arguments is `missing`.

The infix operation `a ⊻ b` is a synonym for `xor(a,b)`, and `⊻` can be typed by tab-completing `\xor` or `\veebar` in the Julia REPL.

Examples

```
julia> xor(true, false)
true

julia> xor(true, true)
false

julia> xor(true, missing)
missing

julia> false ⊻ false
false

julia> [true; true; false] .⊻ [true; false; false]
3-element BitVector:
 0
 1
 0
```

source

Base.nand – Function.

```
nand(x, y)
⊼(x, y)
```

Bitwise nand (not and) of x and y. Implements three-valued logic, returning `missing` if one of the arguments is `missing`.

The infix operation a ⊼ b is a synonym for `nand(a,b)`, and ⊼ can be typed by tab-completing `\nand` or `\barwedge` in the Julia REPL.

Examples

```
julia> nand(true, false)
true

julia> nand(true, true)
false

julia> nand(true, missing)
missing

julia> false ⊼ false
true

julia> [true; true; false] .⊼ [true; false; false]
3-element BitVector:
 0
 1
 1
```

source

Base.nor – Function.

```
nor(x, y)
⊽(x, y)
```

Bitwise nor (not or) of x and y. Implements three-valued logic, returning `missing` if one of the arguments is `missing` and the other is not `true`.

The infix operation a ⊽ b is a synonym for `nor(a,b)`, and ⊽ can be typed by tab-completing `\nor` or `\barvee` in the Julia REPL.

Examples

```
julia> nor(true, false)
false

julia> nor(true, true)
false

julia> nor(true, missing)
false
```

```
julia> false ⊻ false
true

julia> false ⊻ missing
missing

julia> [true; true; false] .⊻ [true; false; false]
3-element BitVector:
 0
 0
 1
```

source

Base.:! – Function.

```
!f::Function
```

Predicate function negation: when the argument of ! is a function, it returns a composed function which computes the boolean negation of f.

See also ∘.

Examples

```
julia> str = "∀ ε > 0, ∃ δ > 0: |x-y| < δ ⇒ |f(x)-f(y)| < ε"
"∀ ε > 0, ∃ δ > 0: |x-y| < δ ⇒ |f(x)-f(y)| < ε"

julia> filter(isletter, str)
"εδxyδfxfyε"

julia> filter(!isletter, str)
"∀  > 0, ∃  > 0: |-| <  ⇒ |()-()| < "
```

Julia 1.9

Starting with Julia 1.9, !f returns a ComposedFunction instead of an anonymous function.

source

```
!(x)
```

Boolean not. Implements three-valued logic, returning missing if x is missing.

See also ~ for bitwise not.

Examples

```
julia> !true
false

julia> !false
true
```

```
julia> !missing
missing

julia> .![true false true]
1×3 BitMatrix:
 0  1  0
```

source

&& – Keyword.

```
x && y
```

Short-circuiting boolean AND.

This is equivalent to `x ? y : false`: it returns `false` if x is `false` and the result of evaluating y if x is `true`. Note that if y is an expression, it is only evaluated when x is `true`, which is called "short-circuiting" behavior.

Also, y does not need to have a boolean value. This means that `(condition) && (statement)` can be used as shorthand for `if condition; statement; end` for an arbitrary `statement`.

See also &, the ternary operator `? :`, and the manual section on control flow.

Examples

```
julia> x = 3;

julia> x > 1 && x < 10 && x isa Int
true

julia> x < 0 && error("expected positive x")
false

julia> x > 0 && "not a boolean"
"not a boolean"
```

source

|| – Keyword.

```
x || y
```

Short-circuiting boolean OR.

This is equivalent to `x ? true : y`: it returns `true` if x is `true` and the result of evaluating y if x is `false`. Note that if y is an expression, it is only evaluated when x is `false`, which is called "short-circuiting" behavior.

Also, y does not need to have a boolean value. This means that `(condition) || (statement)` can be used as shorthand for `if !(condition); statement; end` for an arbitrary `statement`.

See also: |, `xor`, &&.

Examples

```
julia> pi < 3 || □ < 3
true

julia> false || true || println("neither is true!")
true

julia> pi < 3 || "not a boolean"
"not a boolean"
```

source

46.2 Mathematical Functions

Base.isapprox – Function.

```
isapprox(x; kwargs...) / ≈(x; kwargs...)
```

Create a function that compares its argument to x using ≈, i.e. a function equivalent to `y -> y ≈ x`.

The keyword arguments supported here are the same as those in the 2-argument `isapprox`.

Julia 1.5

This method requires Julia 1.5 or later.

source

```
isapprox(x, y; atol::Real=0, rtol::Real=atol>0 ? 0 : √eps, nans::Bool=false[, norm::Function])
```

Inexact equality comparison. Two numbers compare equal if their relative distance *or* their absolute distance is within tolerance bounds: `isapprox` returns `true` if `norm(x-y) <= max(atol, rtol*max(norm(x), norm(y)))`. The default `atol` (absolute tolerance) is zero and the default `rtol` (relative tolerance) depends on the types of x and y. The keyword argument `nans` determines whether or not NaN values are considered equal (defaults to false).

For real or complex floating-point values, if an `atol > 0` is not specified, `rtol` defaults to the square root of `eps` of the type of x or y, whichever is bigger (least precise). This corresponds to requiring equality of about half of the significant digits. Otherwise, e.g. for integer arguments or if an `atol > 0` is supplied, `rtol` defaults to zero.

The `norm` keyword defaults to `abs` for numeric `(x,y)` and to `LinearAlgebra.norm` for arrays (where an alternative `norm` choice is sometimes useful). When x and y are arrays, if `norm(x-y)` is not finite (i.e. `±Inf` or `NaN`), the comparison falls back to checking whether all elements of x and y are approximately equal component-wise.

The binary operator ≈ is equivalent to `isapprox` with the default arguments, and `x ≉ y` is equivalent to `!isapprox(x,y)`.

Note that `x ≈ 0` (i.e., comparing to zero with the default tolerances) is equivalent to `x == 0` since the default `atol` is `0`. In such cases, you should either supply an appropriate `atol` (or use `norm(x) ≤ atol`) or rearrange your code (e.g. use `x ≈ y` rather than `x - y ≈ 0`). It is not possible to pick a nonzero `atol` automatically because it depends on the overall scaling (the "units") of your problem: for example, in

`x - y ≈ 0`, `atol=1e-9` is an absurdly small tolerance if x is the radius of the Earth in meters, but an absurdly large tolerance if x is the radius of a Hydrogen atom in meters.

Julia 1.6

Passing the `norm` keyword argument when comparing numeric (non-array) arguments requires Julia 1.6 or later.

Examples

```
julia> isapprox(0.1, 0.15; atol=0.05)
true

julia> isapprox(0.1, 0.15; rtol=0.34)
true

julia> isapprox(0.1, 0.15; rtol=0.33)
false

julia> 0.1 + 1e-10 ≈ 0.1
true

julia> 1e-10 ≈ 0
false

julia> isapprox(1e-10, 0, atol=1e-8)
true

julia> isapprox([10.0^9, 1.0], [10.0^9, 2.0]) # using `norm`
true
```

source

Base.sin – Method.

```
sin(x::T) where {T <: Number} -> float(T)
```

Compute sine of x, where x is in radians.

Throw a DomainError if `isinf(x)`, return a `T(NaN)` if `isnan(x)`.

See also sind, sinpi, sincos, cis, asin.

Examples

```
julia> round.(sin.(range(0, 2pi, length=9)'), digits=3)
1×9 Matrix{Float64}:
 0.0  0.707  1.0  0.707  0.0  -0.707  -1.0  -0.707  -0.0

julia> sind(45)
0.7071067811865476

julia> sinpi(1/4)
0.7071067811865475
```

```
julia> round.(sincos(pi/6), digits=3)
(0.5, 0.866)

julia> round(cis(pi/6), digits=3)
0.866 + 0.5im

julia> round(exp(im*pi/6), digits=3)
0.866 + 0.5im
```

source

Base.cos – Method.

```
cos(x::T) where {T <: Number} -> float(T)
```

Compute cosine of x, where x is in radians.

Throw a DomainError if isinf(x), return a T(NaN) if isnan(x).

See also cosd, cospi, sincos, cis.

source

Base.Math.sincos – Method.

```
sincos(x::T) where T -> Tuple{float(T),float(T)}
```

Simultaneously compute the sine and cosine of x, where x is in radians, returning a tuple (sine, cosine).

Throw a DomainError if isinf(x), return a (T(NaN), T(NaN)) if isnan(x).

See also cis, sincospi, sincosd.

source

Base.tan – Method.

```
tan(x::T) where {T <: Number} -> float(T)
```

Compute tangent of x, where x is in radians.

Throw a DomainError if isinf(x), return a T(NaN) if isnan(x).

See also tanh.

source

Base.Math.sind – Function.

```
sind(x::T) where T -> float(T)
```

Compute sine of x, where x is in degrees. If x is a matrix, x needs to be a square matrix.

Throw a DomainError if isinf(x), return a T(NaN) if isnan(x).

Julia 1.7

Matrix arguments require Julia 1.7 or later.

source

Base.Math.cosd – Function.

```
cosd(x::T) where T -> float(T)
```

Compute cosine of x, where x is in degrees. If x is a matrix, x needs to be a square matrix.

Throw a DomainError if isinf(x), return a T(NaN) if isnan(x).

Julia 1.7

Matrix arguments require Julia 1.7 or later.

source

Base.Math.tand – Function.

```
tand(x::T) where T -> float(T)
```

Compute tangent of x, where x is in degrees. If x is a matrix, x needs to be a square matrix.

Throw a DomainError if isinf(x), return a T(NaN) if isnan(x).

Julia 1.7

Matrix arguments require Julia 1.7 or later.

source

Base.Math.sincosd – Function.

```
sincosd(x::T) where T -> Tuple{float(T),float(T)}
```

Simultaneously compute the sine and cosine of x, where x is in degrees, returning a tuple (sine, cosine).

Throw a DomainError if isinf(x), return a (T(NaN), T(NaN)) tuple if isnan(x).

Julia 1.3

This function requires at least Julia 1.3.

source

Base.Math.sinpi – Function.

```
sinpi(x::T) where T -> float(T)
```

Compute $\sin(\pi x)$ more accurately than `sin(pi*x)`, especially for large `x`.

Throw a `DomainError` if `isinf(x)`, return a `T(NaN)` if `isnan(x)`.

See also `sind`, `cospi`, `sincospi`.

source

`Base.Math.cospi` – Function.

```
cospi(x::T) where T -> float(T)
```

Compute $\cos(\pi x)$ more accurately than `cos(pi*x)`, especially for large `x`.

Throw a `DomainError` if `isinf(x)`, return a `T(NaN)` if `isnan(x)`.

See also: `cispi`, `sincosd`, `sinpi`.

source

`Base.Math.tanpi` – Function.

```
tanpi(x::T) where T -> float(T)
```

Compute $\tan(\pi x)$ more accurately than `tan(pi*x)`, especially for large `x`.

Throw a `DomainError` if `isinf(x)`, return a `T(NaN)` if `isnan(x)`.

Julia 1.10

This function requires at least Julia 1.10.

See also `tand`, `sinpi`, `cospi`, `sincospi`.

source

`Base.Math.sincospi` – Function.

```
sincospi(x::T) where T -> Tuple{float(T),float(T)}
```

Simultaneously compute `sinpi(x)` and `cospi(x)` (the sine and cosine of π*x, where `x` is in radians), returning a tuple `(sine, cosine)`.

Throw a `DomainError` if `isinf(x)`, return a `(T(NaN), T(NaN))` tuple if `isnan(x)`.

Julia 1.6

This function requires Julia 1.6 or later.

See also: `cispi`, `sincosd`, `sinpi`.

source

`Base.sinh` – Method.

```
sinh(x)
```

Compute hyperbolic sine of x.

See also sin.

source

Base.cosh – Method.

```
cosh(x)
```

Compute hyperbolic cosine of x.

See also cos.

source

Base.tanh – Method.

```
tanh(x)
```

Compute hyperbolic tangent of x.

See also tan, atanh.

Examples

```
julia> tanh.(-3:3f0)  # Here 3f0 isa Float32
7-element Vector{Float32}:
 -0.9950548
 -0.9640276
 -0.7615942
  0.0
  0.7615942
  0.9640276
  0.9950548

julia> tan.(im .* (1:3))
3-element Vector{ComplexF64}:
 0.0 + 0.7615941559557649im
 0.0 + 0.9640275800758169im
 0.0 + 0.9950547536867306im
```

source

Base.asin – Method.

```
asin(x::T) where {T <: Number} -> float(T)
```

Compute the inverse sine of x, where the output is in radians.

Return a T(NaN) if isnan(x).

See also asind for output in degrees.

Examples

```
julia> asin.((0, 1/2, 1))
(0.0, 0.5235987755982989, 1.5707963267948966)

julia> asind.((0, 1/2, 1))
(0.0, 30.000000000000004, 90.0)
```

source

Base.acos – Method.

```
acos(x::T) where {T <: Number} -> float(T)
```

Compute the inverse cosine of x, where the output is in radians.

Return a `T(NaN)` if `isnan(x)`.

source

Base.atan – Method.

```
atan(y)
atan(y, x)
```

Compute the inverse tangent of y or y/x, respectively.

For one real argument, this is the angle in radians between the positive *x*-axis and the point (1, *y*), returning a value in the interval $[-\pi/2, \pi/2]$.

For two arguments, this is the angle in radians between the positive *x*-axis and the point (*x*, *y*), returning a value in the interval $[-\pi, \pi]$. This corresponds to a standard atan2 function. Note that by convention `atan(0.0,x)` is defined as π and `atan(-0.0,x)` is defined as $-\pi$ when `x < 0`.

See also atand for degrees.

Examples

```
julia> rad2deg(atan(-1/√3))
-30.000000000000004

julia> rad2deg(atan(-1, √3))
-30.000000000000004

julia> rad2deg(atan(1, -√3))
150.0
```

source

Base.Math.asind – Function.

```
asind(x)
```

Compute the inverse sine of x, where the output is in degrees. If x is a matrix, x needs to be a square matrix.

Julia 1.7

Matrix arguments require Julia 1.7 or later.

source

Base.Math.acosd – Function.

```
acosd(x)
```

Compute the inverse cosine of x, where the output is in degrees. If x is a matrix, x needs to be a square matrix.

Julia 1.7

Matrix arguments require Julia 1.7 or later.

source

Base.Math.atand – Function.

```
atand(y::T) where T -> float(T)
atand(y::T, x::S) where {T,S} -> promote_type(T,S)
atand(y::AbstractMatrix{T}) where T -> AbstractMatrix{Complex{float(T)}}
```

Compute the inverse tangent of y or y/x, respectively, where the output is in degrees.

Return a NaN if `isnan(y)` or `isnan(x)`. The returned NaN is either a T in the single argument version, or a `promote_type(T,S)` in the two argument version.

Julia 1.7

The one-argument method supports square matrix arguments as of Julia 1.7.

source

Base.Math.sec – Method.

```
sec(x::T) where {T <: Number} -> float(T)
```

Compute the secant of x, where x is in radians.

Throw a DomainError if `isinf(x)`, return a `T(NaN)` if `isnan(x)`.

source

Base.Math.csc – Method.

```
csc(x::T) where {T <: Number} -> float(T)
```

Compute the cosecant of x, where x is in radians.

Throw a DomainError if `isinf(x)`, return a `T(NaN)` if `isnan(x)`.

source

Base.Math.cot – Method.

```
cot(x::T) where {T <: Number} -> float(T)
```

Compute the cotangent of x, where x is in radians.

Throw a DomainError if `isinf(x)`, return a `T(NaN)` if `isnan(x)`.

source

Base.Math.secd – Function.

```
secd(x::T) where {T <: Number} -> float(T)
```

Compute the secant of x, where x is in degrees.

Throw a DomainError if isinf(x), return a T(NaN) if isnan(x).

source

Base.Math.cscd – Function.

```
cscd(x::T) where {T <: Number} -> float(T)
```

Compute the cosecant of x, where x is in degrees.

Throw a DomainError if isinf(x), return a T(NaN) if isnan(x).

source

Base.Math.cotd – Function.

```
cotd(x::T) where {T <: Number} -> float(T)
```

Compute the cotangent of x, where x is in degrees.

Throw a DomainError if isinf(x), return a T(NaN) if isnan(x).

source

Base.Math.asec – Method.

```
asec(x::T) where {T <: Number} -> float(T)
```

Compute the inverse secant of x, where the output is in radians.

source

Base.Math.acsc – Method.

```
acsc(x::T) where {T <: Number} -> float(T)
```

Compute the inverse cosecant of x, where the output is in radians.

source

Base.Math.acot – Method.

```
acot(x::T) where {T <: Number} -> float(T)
```

Compute the inverse cotangent of x, where the output is in radians.

source

Base.Math.asecd – Function.

```
asecd(x)
```

Compute the inverse secant of x, where the output is in degrees. If x is a matrix, x needs to be a square matrix.

Julia 1.7

Matrix arguments require Julia 1.7 or later.

source

Base.Math.acscd – Function.

```
acscd(x)
```

Compute the inverse cosecant of x, where the output is in degrees. If x is a matrix, x needs to be a square matrix.

Julia 1.7

Matrix arguments require Julia 1.7 or later.

source

Base.Math.acotd – Function.

```
acotd(x)
```

Compute the inverse cotangent of x, where the output is in degrees. If x is a matrix, x needs to be a square matrix.

Julia 1.7

Matrix arguments require Julia 1.7 or later.

source

Base.Math.sech – Method.

```
sech(x::T) where {T <: Number} -> float(T)
```

Compute the hyperbolic secant of x.

Return a `T(NaN)` if `isnan(x)`.

source

Base.Math.csch – Method.

```
csch(x::T) where {T <: Number} -> float(T)
```

Compute the hyperbolic cosecant of x.

Return a `T(NaN)` if `isnan(x)`.

source

Base.Math.coth – Method.

```
coth(x::T) where {T <: Number} -> float(T)
```

Compute the hyperbolic cotangent of x.

Return a `T(NaN)` if `isnan(x)`.

source

`Base.asinh` – Method.

```
asinh(x)
```

Compute the inverse hyperbolic sine of x.

source

`Base.acosh` – Method.

```
acosh(x)
```

Compute the inverse hyperbolic cosine of x.

source

`Base.atanh` – Method.

```
atanh(x)
```

Compute the inverse hyperbolic tangent of x.

source

`Base.Math.asech` – Method.

```
asech(x::T) where {T <: Number} -> float(T)
```

Compute the inverse hyperbolic secant of x.

source

`Base.Math.acsch` – Method.

```
acsch(x::T) where {T <: Number} -> float(T)
```

Compute the inverse hyperbolic cosecant of x.

source

`Base.Math.acoth` – Method.

```
acoth(x::T) where {T <: Number} -> float(T)
```

Compute the inverse hyperbolic cotangent of x.

source

`Base.Math.sinc` – Function.

```
sinc(x::T) where {T <: Number} -> float(T)
```

Compute normalized sinc function $\operatorname{sinc}(x) = \sin(\pi x)/(\pi x)$ if $x \neq 0$, and 1 if $x = 0$.

Return a `T(NaN)` if `isnan(x)`.

See also `cosc`, its derivative.

source

`Base.Math.cosc` – Function.

```
cosc(x::T) where {T <: Number} -> float(T)
```

Compute $\cos(\pi x)/x - \sin(\pi x)/(\pi x^2)$ if $x \neq 0$, and 0 if $x = 0$. This is the derivative of `sinc(x)`.

Return a `T(NaN)` if `isnan(x)`.

See also `sinc`.

source

`Base.Math.deg2rad` – Function.

```
deg2rad(x)
```

Convert x from degrees to radians.

See also `rad2deg`, `sind`, `pi`.

Examples

```
julia> deg2rad(90)
1.5707963267948966
```

source

`Base.Math.rad2deg` – Function.

```
rad2deg(x)
```

Convert x from radians to degrees.

See also `deg2rad`.

Examples

```
julia> rad2deg(pi)
180.0
```

source

`Base.Math.hypot` – Function.

```
hypot(x, y)
```

Compute the hypotenuse $\sqrt{|x|^2 + |y|^2}$ avoiding overflow and underflow.

This code is an implementation of the algorithm described in: An Improved Algorithm for `hypot(a,b)` by Carlos F. Borges The article is available online at arXiv at the link https://arxiv.org/abs/1904.09481

```
hypot(x...)
```

Compute the hypotenuse $\sqrt{\sum |x_i|^2}$ avoiding overflow and underflow.

See also `norm` in the `LinearAlgebra` standard library.

Examples

```
julia> a = Int64(10)^10;

julia> hypot(a, a)
1.4142135623730951e10

julia> √(a^2 + a^2) # a^2 overflows
ERROR: DomainError with -2.914184810805068e18:
sqrt was called with a negative real argument but will only return a complex result if called
↪  with a complex argument. Try sqrt(Complex(x)).
Stacktrace:
[...]

julia> hypot(3, 4im)
5.0

julia> hypot(-5.7)
5.7

julia> hypot(3, 4im, 12.0)
13.0

julia> using LinearAlgebra

julia> norm([a, a, a, a]) == hypot(a, a, a, a)
true
```

source

`Base.log` – Method.

```
log(x)
```

Compute the natural logarithm of `x`.

Throw a `DomainError` for negative `Real` arguments. Use `Complex` arguments to obtain `Complex` results.

Branch cut

`log` has a branch cut along the negative real axis; `-0.0im` is taken to be below the axis.

See also , `log1p`, `log2`, `log10`.

Examples

```
julia> log(2)
0.6931471805599453

julia> log(-3)
ERROR: DomainError with -3.0:
log was called with a negative real argument but will only return a complex result if called
↪  with a complex argument. Try log(Complex(x)).
Stacktrace:
 [1] throw_complex_domainerror(::Symbol, ::Float64) at ./math.jl:31
[...]

julia> log(-3 + 0im)
1.0986122886681098 + 3.141592653589793im

julia> log(-3 - 0.0im)
1.0986122886681098 - 3.141592653589793im

julia> log.(exp.(-1:1))
3-element Vector{Float64}:
 -1.0
  0.0
  1.0
```

source

Base.log – Method.

```
log(b,x)
```

Compute the base b logarithm of x. Throw a DomainError for negative Real arguments.

Examples

```
julia> log(4,8)
1.5

julia> log(4,2)
0.5

julia> log(-2, 3)
ERROR: DomainError with -2.0:
log was called with a negative real argument but will only return a complex result if called
↪  with a complex argument. Try log(Complex(x)).
Stacktrace:
 [1] throw_complex_domainerror(::Symbol, ::Float64) at ./math.jl:31
[...]

julia> log(2, -3)
ERROR: DomainError with -3.0:
log was called with a negative real argument but will only return a complex result if called
↪  with a complex argument. Try log(Complex(x)).
Stacktrace:
 [1] throw_complex_domainerror(::Symbol, ::Float64) at ./math.jl:31
[...]
```

Note

If b is a power of 2 or 10, `log2` or `log10` should be used, as these will typically be faster and more accurate. For example,

```
julia> log(100,1000000)
2.9999999999999996

julia> log10(1000000)/2
3.0
```

source

Base.log2 – Function.

```
log2(x)
```

Compute the logarithm of x to base 2. Throw a `DomainError` for negative `Real` arguments.

See also: `exp2`, `ldexp`, `ispow2`.

Examples

```
julia> log2(4)
2.0

julia> log2(10)
3.321928094887362

julia> log2(-2)
ERROR: DomainError with -2.0:
log2 was called with a negative real argument but will only return a complex result if called
↪  with a complex argument. Try log2(Complex(x)).
Stacktrace:
 [1] throw_complex_domainerror(f::Symbol, x::Float64) at ./math.jl:31
[...]

julia> log2.(2.0 .^ (-1:1))
3-element Vector{Float64}:
 -1.0
  0.0
  1.0
```

source

Base.log10 – Function.

```
log10(x)
```

Compute the logarithm of x to base 10. Throw a `DomainError` for negative `Real` arguments.

Examples

```
julia> log10(100)
2.0

julia> log10(2)
0.3010299956639812

julia> log10(-2)
ERROR: DomainError with -2.0:
log10 was called with a negative real argument but will only return a complex result if called
↪  with a complex argument. Try log10(Complex(x)).
Stacktrace:
 [1] throw_complex_domainerror(f::Symbol, x::Float64) at ./math.jl:31
[...]
```

source

Base.log1p – Function.

```
log1p(x)
```

Accurate natural logarithm of 1+x. Throw a DomainError for Real arguments less than -1.

Examples

```
julia> log1p(-0.5)
-0.6931471805599453

julia> log1p(0)
0.0

julia> log1p(-2)
ERROR: DomainError with -2.0:
log1p was called with a real argument < -1 but will only return a complex result if called
↪  with a complex argument. Try log1p(Complex(x)).
Stacktrace:
 [1] throw_complex_domainerror(::Symbol, ::Float64) at ./math.jl:31
[...]
```

source

Base.Math.frexp – Function.

```
frexp(val)
```

Return (x,exp) such that x has a magnitude in the interval $[1/2, 1)$ or 0, and val is equal to $x \times 2^{exp}$.

See also significand, exponent, ldexp.

Examples

```
julia> frexp(6.0)
(0.75, 3)

julia> significand(6.0), exponent(6.0)  # interval [1, 2) instead
(1.5, 2)
```

```
julia> frexp(0.0), frexp(NaN), frexp(-Inf)  # exponent would give an error
((0.0, 0), (NaN, 0), (-Inf, 0))
```

source

Base.exp – Method.

```
exp(x)
```

Compute the natural base exponential of x, in other words ^{x}.

See also exp2, exp10 and cis.

Examples

```
julia> exp(1.0)
2.718281828459045

julia> exp(im * pi) ≈ cis(pi)
true
```

source

Base.exp2 – Function.

```
exp2(x)
```

Compute the base 2 exponential of x, in other words 2^x.

See also ldexp, <<.

Examples

```
julia> exp2(5)
32.0

julia> 2^5
32

julia> exp2(63) > typemax(Int)
true
```

source

Base.exp10 – Function.

```
exp10(x)
```

Compute the base 10 exponential of x, in other words 10^x.

Examples

```
julia> exp10(2)
100.0

julia> 10^2
100
```

source

Base.Math.ldexp – Function.

```
ldexp(x, n)
```

Compute $x \times 2^n$.

See also frexp, exponent.

Examples

```
julia> ldexp(5.0, 2)
20.0
```

source

Base.Math.modf – Function.

```
modf(x)
```

Return a tuple (fpart, ipart) of the fractional and integral parts of a number. Both parts have the same sign as the argument.

Examples

```
julia> modf(3.5)
(0.5, 3.0)

julia> modf(-3.5)
(-0.5, -3.0)
```

source

Base.expm1 – Function.

```
expm1(x)
```

Accurately compute $e^x - 1$. It avoids the loss of precision involved in the direct evaluation of exp(x) - 1 for small values of x.

Examples

```
julia> expm1(1e-16)
1.0e-16

julia> exp(1e-16) - 1
0.0
```

source

Base.round – Function.

```
round([T,] x, [r::RoundingMode])
round(x, [r::RoundingMode]; digits::Integer=0, base = 10)
round(x, [r::RoundingMode]; sigdigits::Integer, base = 10)
```

Rounds the number x.

Without keyword arguments, x is rounded to an integer value, returning a value of type T, or of the same type of x if no T is provided. An `InexactError` will be thrown if the value is not representable by T, similar to `convert`.

If the `digits` keyword argument is provided, it rounds to the specified number of digits after the decimal place (or before if `digits` is negative), in base `base`.

If the `sigdigits` keyword argument is provided, it rounds to the specified number of significant digits, in base `base`.

The `RoundingMode` `r` controls the direction of the rounding; the default is `RoundNearest`, which rounds to the nearest integer, with ties (fractional values of 0.5) being rounded to the nearest even integer. Note that `round` may give incorrect results if the global rounding mode is changed (see `rounding`).

When rounding to a floating point type, will round to integers representable by that type (and Inf) rather than true integers. Inf is treated as one ulp greater than the `floatmax(T)` for purposes of determining "nearest", similar to `convert`.

Examples

```
julia> round(1.7)
2.0

julia> round(Int, 1.7)
2

julia> round(1.5)
2.0

julia> round(2.5)
2.0

julia> round(pi; digits=2)
3.14

julia> round(pi; digits=3, base=2)
3.125

julia> round(123.456; sigdigits=2)
120.0

julia> round(357.913; sigdigits=4, base=2)
352.0

julia> round(Float16, typemax(UInt128))
Inf16

julia> floor(Float16, typemax(UInt128))
Float16(6.55e4)
```

Note

Rounding to specified digits in bases other than 2 can be inexact when operating on binary floating point numbers. For example, the `Float64` value represented by `1.15` is actually *less* than 1.15, yet will be rounded to 1.2. For example:

```
julia> x = 1.15
1.15

julia> big(1.15)
1.149999999999999911182158029987476766109466552734375

julia> x < 115//100
true

julia> round(x, digits=1)
1.2
```

Extensions

To extend `round` to new numeric types, it is typically sufficient to define `Base.round(x::NewType, r::RoundingMode)`.

source

`Base.Rounding.RoundingMode` – Type.

```
RoundingMode
```

A type used for controlling the rounding mode of floating point operations (via `rounding`/`setrounding` functions), or as optional arguments for rounding to the nearest integer (via the `round` function).

Currently supported rounding modes are:

- `RoundNearest` (default)
- `RoundNearestTiesAway`
- `RoundNearestTiesUp`
- `RoundToZero`
- `RoundFromZero`
- `RoundUp`
- `RoundDown`

Julia 1.9

`RoundFromZero` requires at least Julia 1.9. Prior versions support `RoundFromZero` for `BigFloats` only.

source

`Base.Rounding.RoundNearest` – Constant.

```
RoundNearest
```

The default rounding mode. Rounds to the nearest integer, with ties (fractional values of 0.5) being rounded to the nearest even integer.

source

Base.Rounding.RoundNearestTiesAway – Constant.

```
RoundNearestTiesAway
```

Rounds to nearest integer, with ties rounded away from zero (C/C++ `round` behaviour).

source

Base.Rounding.RoundNearestTiesUp – Constant.

```
RoundNearestTiesUp
```

Rounds to nearest integer, with ties rounded toward positive infinity (Java/JavaScript `round` behaviour).

source

Base.Rounding.RoundToZero – Constant.

```
RoundToZero
```

`round` using this rounding mode is an alias for `trunc`.

source

Base.Rounding.RoundFromZero – Constant.

```
RoundFromZero
```

Rounds away from zero.

Julia 1.9

`RoundFromZero` requires at least Julia 1.9. Prior versions support `RoundFromZero` for `BigFloats` only.

Examples

```
julia> BigFloat("1.0000000000000001", 5, RoundFromZero)
1.06
```

source

Base.Rounding.RoundUp – Constant.

```
RoundUp
```

`round` using this rounding mode is an alias for `ceil`.

source

Base.Rounding.RoundDown – Constant.

```
RoundDown
```

round using this rounding mode is an alias for floor.

source

Base.round – Method.

```
round(z::Complex[, RoundingModeReal, [RoundingModeImaginary]])
round(z::Complex[, RoundingModeReal, [RoundingModeImaginary]]; digits=0, base=10)
round(z::Complex[, RoundingModeReal, [RoundingModeImaginary]]; sigdigits, base=10)
```

Return the nearest integral value of the same type as the complex-valued z to z, breaking ties using the specified RoundingModes. The first RoundingMode is used for rounding the real components while the second is used for rounding the imaginary components.

RoundingModeReal and RoundingModeImaginary default to RoundNearest, which rounds to the nearest integer, with ties (fractional values of 0.5) being rounded to the nearest even integer.

Examples

```
julia> round(3.14 + 4.5im)
3.0 + 4.0im

julia> round(3.14 + 4.5im, RoundUp, RoundNearestTiesUp)
4.0 + 5.0im

julia> round(3.14159 + 4.512im; digits = 1)
3.1 + 4.5im

julia> round(3.14159 + 4.512im; sigdigits = 3)
3.14 + 4.51im
```

source

Base.ceil – Function.

```
ceil([T,] x)
ceil(x; digits::Integer= [, base = 10])
ceil(x; sigdigits::Integer= [, base = 10])
```

ceil(x) returns the nearest integral value of the same type as x that is greater than or equal to x.

ceil(T, x) converts the result to type T, throwing an InexactError if the ceiled value is not representable as a T.

Keywords digits, sigdigits and base work as for round.

To support ceil for a new type, define Base.round(x::NewType, ::RoundingMode{:Up}).

source

Base.floor – Function.

```
floor([T,] x)
floor(x; digits::Integer= [, base = 10])
floor(x; sigdigits::Integer= [, base = 10])
```

`floor(x)` returns the nearest integral value of the same type as `x` that is less than or equal to `x`.

`floor(T, x)` converts the result to type `T`, throwing an `InexactError` if the floored value is not representable a `T`.

Keywords `digits`, `sigdigits` and `base` work as for `round`.

To support `floor` for a new type, define `Base.round(x::NewType, ::RoundingMode{:Down})`.

source

`Base.trunc` – Function.

```
trunc([T,] x)
trunc(x; digits::Integer= [, base = 10])
trunc(x; sigdigits::Integer= [, base = 10])
```

`trunc(x)` returns the nearest integral value of the same type as `x` whose absolute value is less than or equal to the absolute value of `x`.

`trunc(T, x)` converts the result to type `T`, throwing an `InexactError` if the truncated value is not representable a `T`.

Keywords `digits`, `sigdigits` and `base` work as for `round`.

To support `trunc` for a new type, define `Base.round(x::NewType, ::RoundingMode{:ToZero})`.

See also: `%`, `floor`, `unsigned`, `unsafe_trunc`.

Examples

```
julia> trunc(2.22)
2.0

julia> trunc(-2.22, digits=1)
-2.2

julia> trunc(Int, -2.22)
-2
```

source

`Base.unsafe_trunc` – Function.

```
unsafe_trunc(T, x)
```

Return the nearest integral value of type `T` whose absolute value is less than or equal to the absolute value of `x`. If the value is not representable by `T`, an arbitrary value will be returned. See also `trunc`.

Examples

```
julia> unsafe_trunc(Int, -2.2)
-2

julia> unsafe_trunc(Int, NaN) isa Int
true
```

source

Base.min – Function.

```
min(x, y, ...)
```

Return the minimum of the arguments, with respect to `isless`. If any of the arguments is `missing`, return `missing`. See also the `minimum` function to take the minimum element from a collection.

Examples

```
julia> min(2, 5, 1)
1

julia> min(4, missing, 6)
missing
```

source

Base.max – Function.

```
max(x, y, ...)
```

Return the maximum of the arguments, with respect to `isless`. If any of the arguments is `missing`, return `missing`. See also the `maximum` function to take the maximum element from a collection.

Examples

```
julia> max(2, 5, 1)
5

julia> max(5, missing, 6)
missing
```

source

Base.minmax – Function.

```
minmax(x, y)
```

Return `(min(x,y), max(x,y))`.

See also `extrema` that returns `(minimum(x), maximum(x))`.

Examples

```
julia> minmax('c','b')
('b', 'c')
```

source

Base.clamp – Function.

```
clamp(x::Integer, r::AbstractUnitRange)
```

Clamp x to lie within range r.

Julia 1.6

This method requires at least Julia 1.6.

source

```
clamp(x, T)::T
```

Clamp x between typemin(T) and typemax(T) and convert the result to type T.

See also trunc.

Examples

```
julia> clamp(200, Int8)
127

julia> clamp(-200, Int8)
-128

julia> trunc(Int, 4pi^2)
39
```

source

```
clamp(x, lo, hi)
```

Return x if lo <= x <= hi. If x > hi, return hi. If x < lo, return lo. Arguments are promoted to a common type.

See also clamp!, min, max.

Julia 1.3

missing as the first argument requires at least Julia 1.3.

Examples

```
julia> clamp.([pi, 1.0, big(10)], 2.0, 9.0)
3-element Vector{BigFloat}:
 3.14159265358979323846264338327950288419716939937510582097494459230781640628619 8
 2.0
 9.0

julia> clamp.([11, 8, 5], 10, 6)  # an example where lo > hi
3-element Vector{Int64}:
```

```
 6
 6
10
```

source

Base.clamp! – Function.

```
clamp!(array::AbstractArray, lo, hi)
```

Restrict values in array to the specified range, in-place. See also clamp.

Julia 1.3

missing entries in array require at least Julia 1.3.

Examples

```
julia> row = collect(-4:4)';

julia> clamp!(row, 0, Inf)
1×9 adjoint(::Vector{Int64}) with eltype Int64:
 0  0  0  0  0  1  2  3  4

julia> clamp.((-4:4)', 0, Inf)
1×9 Matrix{Float64}:
 0.0  0.0  0.0  0.0  0.0  1.0  2.0  3.0  4.0
```

source

Base.abs – Function.

```
abs(x)
```

The absolute value of x.

When abs is applied to signed integers, overflow may occur, resulting in the return of a negative value. This overflow occurs only when abs is applied to the minimum representable value of a signed integer. That is, when x == typemin(typeof(x)), abs(x) == x < 0, not -x as might be expected.

See also: abs2, unsigned, sign.

Examples

```
julia> abs(-3)
3

julia> abs(1 + im)
1.4142135623730951

julia> abs.(Int8[-128 -127 -126 0 126 127])  # overflow at typemin(Int8)
1×6 Matrix{Int8}:
 -128  127  126  0  126  127

julia> maximum(abs, [1, -2, 3, -4])
4
```

source

Base.Checked – Module.

```
Checked
```

The Checked module provides arithmetic functions for the built-in signed and unsigned Integer types which throw an error when an overflow occurs. They are named like checked_sub, checked_div, etc. In addition, add_with_overflow, sub_with_overflow, mul_with_overflow return both the unchecked results and a boolean value denoting the presence of an overflow.

source

Base.Checked.checked_abs – Function.

```
Base.checked_abs(x)
```

Calculates abs(x), checking for overflow errors where applicable. For example, standard two's complement signed integers (e.g. Int) cannot represent abs(typemin(Int)), thus leading to an overflow.

The overflow protection may impose a perceptible performance penalty.

source

Base.Checked.checked_neg – Function.

```
Base.checked_neg(x)
```

Calculates -x, checking for overflow errors where applicable. For example, standard two's complement signed integers (e.g. Int) cannot represent -typemin(Int), thus leading to an overflow.

The overflow protection may impose a perceptible performance penalty.

source

Base.Checked.checked_add – Function.

```
Base.checked_add(x, y)
```

Calculates x+y, checking for overflow errors where applicable.

The overflow protection may impose a perceptible performance penalty.

source

Base.Checked.checked_sub – Function.

```
Base.checked_sub(x, y)
```

Calculates x-y, checking for overflow errors where applicable.

The overflow protection may impose a perceptible performance penalty.

source

Base.Checked.checked_mul – Function.

```
Base.checked_mul(x, y)
```

Calculates x*y, checking for overflow errors where applicable.

The overflow protection may impose a perceptible performance penalty.

source

`Base.Checked.checked_div` – Function.

```
Base.checked_div(x, y)
```

Calculates `div(x,y)`, checking for overflow errors where applicable.

The overflow protection may impose a perceptible performance penalty.

source

`Base.Checked.checked_rem` – Function.

```
Base.checked_rem(x, y)
```

Calculates x%y, checking for overflow errors where applicable.

The overflow protection may impose a perceptible performance penalty.

source

`Base.Checked.checked_fld` – Function.

```
Base.checked_fld(x, y)
```

Calculates `fld(x,y)`, checking for overflow errors where applicable.

The overflow protection may impose a perceptible performance penalty.

source

`Base.Checked.checked_mod` – Function.

```
Base.checked_mod(x, y)
```

Calculates `mod(x,y)`, checking for overflow errors where applicable.

The overflow protection may impose a perceptible performance penalty.

source

`Base.Checked.checked_cld` – Function.

```
Base.checked_cld(x, y)
```

Calculates `cld(x,y)`, checking for overflow errors where applicable.

The overflow protection may impose a perceptible performance penalty.

source

`Base.Checked.checked_pow` – Function.

```
Base.checked_pow(x, y)
```

Calculates `^(x,y)`, checking for overflow errors where applicable.

The overflow protection may impose a perceptible performance penalty.

source

`Base.Checked.add_with_overflow` – Function.

```
Base.add_with_overflow(x, y) -> (r, f)
```

Calculates `r = x+y`, with the flag `f` indicating whether overflow has occurred.

source

`Base.Checked.sub_with_overflow` – Function.

```
Base.sub_with_overflow(x, y) -> (r, f)
```

Calculates `r = x-y`, with the flag `f` indicating whether overflow has occurred.

source

`Base.Checked.mul_with_overflow` – Function.

```
Base.mul_with_overflow(x, y) -> (r, f)
```

Calculates `r = x*y`, with the flag `f` indicating whether overflow has occurred.

source

`Base.abs2` – Function.

```
abs2(x)
```

Squared absolute value of `x`.

This can be faster than `abs(x)^2`, especially for complex numbers where `abs(x)` requires a square root via `hypot`.

See also `abs`, `conj`, `real`.

Examples

```
julia> abs2(-3)
9

julia> abs2(3.0 + 4.0im)
25.0

julia> sum(abs2, [1+2im, 3+4im])  # LinearAlgebra.norm(x)^2
30
```

source

`Base.copysign` – Function.

```
copysign(x, y) -> z
```

Return z which has the magnitude of x and the same sign as y.

Examples

```
julia> copysign(1, -2)
-1

julia> copysign(-1, 2)
1
```

source

Base.sign – Function.

```
sign(x)
```

Return zero if x==0 and $x/|x|$ otherwise (i.e., ±1 for real x).

See also signbit, zero, copysign, flipsign.

Examples

```
julia> sign(-4.0)
-1.0

julia> sign(99)
1

julia> sign(-0.0)
-0.0

julia> sign(0 + im)
0.0 + 1.0im
```

source

Base.signbit – Function.

```
signbit(x)
```

Return true if the value of the sign of x is negative, otherwise false.

See also sign and copysign.

Examples

```
julia> signbit(-4)
true

julia> signbit(5)
false

julia> signbit(5.5)
false

julia> signbit(-4.1)
true
```

source

Base.flipsign – Function.

```
flipsign(x, y)
```

Return x with its sign flipped if y is negative. For example `abs(x) = flipsign(x,x)`.

Examples

```
julia> flipsign(5, 3)
5

julia> flipsign(5, -3)
-5
```

source

Base.sqrt – Method.

```
sqrt(x)
```

Return $\sqrt{x}$.

Throw a `DomainError` for negative `Real` arguments. Use `Complex` negative arguments instead to obtain a `Complex` result.

The prefix operator √ is equivalent to `sqrt`.

Branch cut

`sqrt` has a branch cut along the negative real axis; `-0.0im` is taken to be below the axis.

See also: `hypot`.

Examples

```
julia> sqrt(big(81))
9.0

julia> sqrt(big(-81))
ERROR: DomainError with -81.0:
NaN result for non-NaN input.
Stacktrace:
 [1] sqrt(::BigFloat) at ./mpfr.jl:501
[...]

julia> sqrt(big(complex(-81)))
0.0 + 9.0im

julia> sqrt(-81 - 0.0im)  # -0.0im is below the branch cut
0.0 - 9.0im

julia> .√(1:4)
4-element Vector{Float64}:
```

```
1.0
1.4142135623730951
1.7320508075688772
2.0
```

source

Base.isqrt – Function.

```
isqrt(n::Integer)
```

Integer square root: the largest integer `m` such that `m*m <= n`.

```
julia> isqrt(5)
2
```

source

Base.Math.cbrt – Method.

```
cbrt(x::Real)
```

Return the cube root of `x`, i.e. $x^{1/3}$. Negative values are accepted (returning the negative real root when $x < 0$).

The prefix operator ∛ is equivalent to `cbrt`.

Examples

```
julia> cbrt(big(27))
3.0

julia> cbrt(big(-27))
-3.0
```

source

Base.Math.fourthroot – Method.

```
fourthroot(x)
```

Return the fourth root of `x` by applying `sqrt` twice successively.

source

Base.real – Function.

```
real(A::AbstractArray)
```

Return an array containing the real part of each entry in array `A`.

Equivalent to `real.(A)`, except that when `eltype(A) <: Real` `A` is returned without copying, and that when `A` has zero dimensions, a 0-dimensional array is returned (rather than a scalar).

Examples

```
julia> real([1, 2im, 3 + 4im])
3-element Vector{Int64}:
 1
 0
 3

julia> real(fill(2 - im))
0-dimensional Array{Int64, 0}:
2
```

source

```
real(T::Type)
```

Return the type that represents the real part of a value of type `T`. e.g: for `T == Complex{R}`, returns `R`. Equivalent to `typeof(real(zero(T)))`.

Examples

```
julia> real(Complex{Int})
Int64

julia> real(Float64)
Float64
```

source

```
real(z)
```

Return the real part of the complex number `z`.

See also: `imag`, `reim`, `complex`, `isreal`, `Real`.

Examples

```
julia> real(1 + 3im)
1
```

source

`Base.imag` – Function.

```
imag(A::AbstractArray)
```

Return an array containing the imaginary part of each entry in array `A`.

Equivalent to `imag.(A)`, except that when `A` has zero dimensions, a 0-dimensional array is returned (rather than a scalar).

Examples

```
julia> imag([1, 2im, 3 + 4im])
3-element Vector{Int64}:
 0
```

```
 2
 4

julia> imag(fill(2 - im))
0-dimensional Array{Int64, 0}:
-1
```

source

```
imag(z)
```

Return the imaginary part of the complex number `z`.

See also: `conj`, `reim`, `adjoint`, `angle`.

Examples

```
julia> imag(1 + 3im)
3
```

source

`Base.reim` – Function.

```
reim(A::AbstractArray)
```

Return a tuple of two arrays containing respectively the real and the imaginary part of each entry in `A`.

Equivalent to `(real.(A), imag.(A))`, except that when `eltype(A) <: Real` `A` is returned without copying to represent the real part, and that when `A` has zero dimensions, a 0-dimensional array is returned (rather than a scalar).

Examples

```
julia> reim([1, 2im, 3 + 4im])
([1, 0, 3], [0, 2, 4])

julia> reim(fill(2 - im))
(fill(2), fill(-1))
```

source

```
reim(z)
```

Return a tuple of the real and imaginary parts of the complex number `z`.

Examples

```
julia> reim(1 + 3im)
(1, 3)
```

source

`Base.conj` – Function.

```
conj(A::AbstractArray)
```

Return an array containing the complex conjugate of each entry in array A.

Equivalent to `conj.(A)`, except that when `eltype(A) <: Real` A is returned without copying, and that when A has zero dimensions, a 0-dimensional array is returned (rather than a scalar).

Examples

```
julia> conj([1, 2im, 3 + 4im])
3-element Vector{Complex{Int64}}:
 1 + 0im
 0 - 2im
 3 - 4im

julia> conj(fill(2 - im))
0-dimensional Array{Complex{Int64}, 0}:
2 + 1im
```

source

```
conj(z)
```

Compute the complex conjugate of a complex number z.

See also: angle, adjoint.

Examples

```
julia> conj(1 + 3im)
1 - 3im
```

source

`Base.angle` – Function.

```
angle(z)
```

Compute the phase angle in radians of a complex number z.

Returns a number `-pi ≤ angle(z) ≤ pi`, and is thus discontinuous along the negative real axis.

See also: atan, cis, rad2deg.

Examples

```
julia> rad2deg(angle(1 + im))
45.0

julia> rad2deg(angle(1 - im))
-45.0

julia> rad2deg(angle(-1 + 1e-20im))
180.0

julia> rad2deg(angle(-1 - 1e-20im))
-180.0
```

source

Base.cis – Function.

```
cis(x)
```

More efficient method for `exp(im*x)` by using Euler's formula: $\cos(x) + i\sin(x) = \exp(ix)$.

See also `cispi`, `sincos`, `exp`, `angle`.

Examples

```
julia> cis(π) ≈ -1
true
```

source

Base.cispi – Function.

```
cispi(x)
```

More accurate method for `cis(pi*x)` (especially for large `x`).

See also `cis`, `sincospi`, `exp`, `angle`.

Examples

```
julia> cispi(10000)
1.0 + 0.0im

julia> cispi(0.25 + 1im)
0.030556854645954562 + 0.03055685464595456im
```

Julia 1.6

This function requires Julia 1.6 or later.

source

Base.binomial – Function.

```
binomial(x::Number, k::Integer)
```

The generalized binomial coefficient, defined for `k ≥ 0` by the polynomial

$$\frac{1}{k!}\prod_{j=0}^{k-1}(x-j)$$

When `k < 0` it returns zero.

For the case of integer `x`, this is equivalent to the ordinary integer binomial coefficient

$$\binom{n}{k} = \frac{n!}{k!(n-k)!}$$

Further generalizations to non-integer k are mathematically possible, but involve the Gamma function and/or the beta function, which are not provided by the Julia standard library but are available in external packages such as SpecialFunctions.jl.

External links

- Binomial coefficient on Wikipedia.

source

```
binomial(n::Integer, k::Integer)
```

The *binomial coefficient* $\binom{n}{k}$, being the coefficient of the kth term in the polynomial expansion of $(1+x)^n$.

If n is non-negative, then it is the number of ways to choose k out of n items:

$$\binom{n}{k} = \frac{n!}{k!(n-k)!}$$

where $n!$ is the `factorial` function.

If n is negative, then it is defined in terms of the identity

$$\binom{n}{k} = (-1)^k \binom{k-n-1}{k}$$

See also `factorial`.

Examples

```
julia> binomial(5, 3)
10

julia> factorial(5) ÷ (factorial(5-3) * factorial(3))
10

julia> binomial(-5, 3)
-35
```

External links

- Binomial coefficient on Wikipedia.

source

`Base.factorial` – Function.

```
factorial(n::Integer)
```

Factorial of n. If n is an `Integer`, the factorial is computed as an integer (promoted to at least 64 bits). Note that this may overflow if n is not small, but you can use `factorial(big(n))` to compute the result exactly in arbitrary precision.

See also `binomial`.

Examples

```
julia> factorial(6)
720

julia> factorial(21)
ERROR: OverflowError: 21 is too large to look up in the table; consider using
↪  `factorial(big(21))` instead
Stacktrace:
[...]

julia> factorial(big(21))
51090942171709440000
```

External links

- Factorial on Wikipedia.

source

`Base.gcd` – Function.

```
gcd(x, y...)
```

Greatest common (positive) divisor (or zero if all arguments are zero). The arguments may be integer and rational numbers.

a is a divisor of b if there exists an integer m such that $ma = b$.

Julia 1.4

Rational arguments require Julia 1.4 or later.

Examples

```
julia> gcd(6, 9)
3

julia> gcd(6, -9)
3

julia> gcd(6, 0)
6

julia> gcd(0, 0)
0
```

```
julia> gcd(1//3, 2//3)
1//3

julia> gcd(1//3, -2//3)
1//3

julia> gcd(1//3, 2)
1//3

julia> gcd(0, 0, 10, 15)
5
```

source

Base.lcm – Function.

```
lcm(x, y...)
```

Least common (positive) multiple (or zero if any argument is zero). The arguments may be integer and rational numbers.

a is a multiple of b if there exists an integer m such that $a = mb$.

Julia 1.4

Rational arguments require Julia 1.4 or later.

Examples

```
julia> lcm(2, 3)
6

julia> lcm(-2, 3)
6

julia> lcm(0, 3)
0

julia> lcm(0, 0)
0

julia> lcm(1//3, 2//3)
2//3

julia> lcm(1//3, -2//3)
2//3

julia> lcm(1//3, 2)
2//1

julia> lcm(1, 3, 5, 7)
105
```

source

Base.gcdx – Function.

```
gcdx(a, b...)
```

Computes the greatest common (positive) divisor of a and b and their Bézout coefficients, i.e. the integer coefficients u and v that satisfy $u*a+v*b = d = gcd(a,b)$. $gcdx(a,b)$ returns (d,u,v).

For more arguments than two, i.e., `gcdx(a, b, c, ...)` the Bézout coefficients are computed recursively, returning a solution `(d, u, v, w, ...)` to $u*a+v*b+w*c+... = d = gcd(a,b,c,...)$.

The arguments may be integer and rational numbers.

Julia 1.4

Rational arguments require Julia 1.4 or later.

Julia 1.12

More or fewer arguments than two require Julia 1.12 or later.

Examples

```
julia> gcdx(12, 42)
(6, -3, 1)

julia> gcdx(240, 46)
(2, -9, 47)

julia> gcdx(15, 12, 20)
(1, 7, -7, -1)
```

Note

Bézout coefficients are *not* uniquely defined. `gcdx` returns the minimal Bézout coefficients that are computed by the extended Euclidean algorithm. (Ref: D. Knuth, TAoCP, 2/e, p. 325, Algorithm X.) For signed integers, these coefficients u and v are minimal in the sense that $|u| < |b/d|$ and $|v| < |a/d|$. Furthermore, the signs of u and v are chosen so that d is positive. For unsigned integers, the coefficients u and v might be near their `typemax`, and the identity then holds only via the unsigned integers' modulo arithmetic.

source

Base.ispow2 – Function.

```
ispow2(n::Number)::Bool
```

Test whether n is an integer power of two.

See also `count_ones`, `prevpow`, `nextpow`.

Examples

```
julia> ispow2(4)
true

julia> ispow2(5)
false

julia> ispow2(4.5)
false

julia> ispow2(0.25)
true

julia> ispow2(1//8)
true
```

Julia 1.6

Support for non-`Integer` arguments was added in Julia 1.6.

source

`Base.nextpow` – Function.

```
nextpow(a, x)
```

The smallest a^n not less than x, where n is a non-negative integer. a must be greater than 1, and x must be greater than 0.

See also `prevpow`.

Examples

```
julia> nextpow(2, 7)
8

julia> nextpow(2, 9)
16

julia> nextpow(5, 20)
25

julia> nextpow(4, 16)
16
```

source

`Base.prevpow` – Function.

```
prevpow(a, x)
```

The largest a^n not greater than x, where n is a non-negative integer. a must be greater than 1, and x must not be less than 1.

See also `nextpow`, `isqrt`.

Examples

```
julia> prevpow(2, 7)
4

julia> prevpow(2, 9)
8

julia> prevpow(5, 20)
5

julia> prevpow(4, 16)
16
```

source

Base.nextprod – Function.

```
nextprod(factors::Union{Tuple,AbstractVector}, n)
```

Next integer greater than or equal to n that can be written as $\prod k_i^{p_i}$ for integers p_1, p_2, etcetera, for factors k_i in `factors`.

Examples

```
julia> nextprod((2, 3), 105)
108

julia> 2^2 * 3^3
108
```

Julia 1.6

The method that accepts a tuple requires Julia 1.6 or later.

source

Base.invmod – Function.

```
invmod(n::Integer, T) where {T <: Base.BitInteger}
invmod(n::T) where {T <: Base.BitInteger}
```

Compute the modular inverse of n in the integer ring of type T, i.e. modulo 2^N where `N = 8*sizeof(T)` (e.g. `N = 32` for `Int32`). In other words, these methods satisfy the following identities:

```
n * invmod(n) == 1
(n * invmod(n, T)) % T == 1
(n % T) * invmod(n, T) == 1
```

Note that * here is modular multiplication in the integer ring, T. This will throw an error if n is even, because then it is not relatively prime with 2^N and thus has no such inverse.

Specifying the modulus implied by an integer type as an explicit value is often inconvenient since the modulus is by definition too big to be represented by the type.

The modular inverse is computed much more efficiently than the general case using the algorithm described in https://arxiv.org/pdf/2204.04342.pdf.

Julia 1.11

The `invmod(n)` and `invmod(n, T)` methods require Julia 1.11 or later.

source

```
invmod(n::Integer, m::Integer)
```

Take the inverse of `n` modulo `m`: `y` such that $ny = 1 \pmod m$, and $div(y, m) = 0$. This will throw an error if $m = 0$, or if $gcd(n, m) \neq 1$.

Examples

```
julia> invmod(2, 5)
3

julia> invmod(2, 3)
2

julia> invmod(5, 6)
5
```

source

`Base.powermod` – Function.

```
powermod(x::Integer, p::Integer, m)
```

Compute $x^p \pmod m$.

Examples

```
julia> powermod(2, 6, 5)
4

julia> mod(2^6, 5)
4

julia> powermod(5, 2, 20)
5

julia> powermod(5, 2, 19)
6

julia> powermod(5, 3, 19)
11
```

source

`Base.ndigits` – Function.

```
ndigits(n::Integer; base::Integer=10, pad::Integer=1)
```

Compute the number of digits in integer n written in base base (base must not be in [-1, 0, 1]), optionally padded with zeros to a specified size (the result will never be less than pad).

See also digits, count_ones.

Examples

```
julia> ndigits(0)
1

julia> ndigits(12345)
5

julia> ndigits(1022, base=16)
3

julia> string(1022, base=16)
"3fe"

julia> ndigits(123, pad=5)
5

julia> ndigits(-123)
3
```

source

Base.add_sum – Function.

```
Base.add_sum(x, y)
```

The reduction operator used in sum. The main difference from + is that small integers are promoted to Int/UInt.

source

Base.uabs – Function.

```
Base.uabs(x::Integer)
```

Return the absolute value of x, possibly returning a different type should the operation be susceptible to overflow. This typically arises when x is a two's complement signed integer, so that abs(typemin(x)) == typemin(x) < 0, in which case the result of uabs(x) will be an unsigned integer of the same size.

source

Base.widemul – Function.

```
widemul(x, y)
```

Multiply x and y, giving the result as a larger type.

See also promote, Base.add_sum.

Examples

```
julia> widemul(Float32(3.0), 4.0) isa BigFloat
true

julia> typemax(Int8) * typemax(Int8)
1

julia> widemul(typemax(Int8), typemax(Int8))  # == 127^2
16129
```

source

Base.Math.evalpoly – Function.

```
evalpoly(x, p)
```

Evaluate the polynomial $\sum_k x^{k-1}p[k]$ for the coefficients p[1], p[2], ...; that is, the coefficients are given in ascending order by power of x. Loops are unrolled at compile time if the number of coefficients is statically known, i.e. when p is a Tuple. This function generates efficient code using Horner's method if x is real, or using a Goertzel-like [1] algorithm if x is complex.

Julia 1.4

This function requires Julia 1.4 or later.

Examples

```
julia> evalpoly(2, (1, 2, 3))
17
```

source

Base.Math.@evalpoly – Macro.

```
@evalpoly(z, c...)
```

Evaluate the polynomial $\sum_k z^{k-1}c[k]$ for the coefficients c[1], c[2], ...; that is, the coefficients are given in ascending order by power of z. This macro expands to efficient inline code that uses either Horner's method or, for complex z, a more efficient Goertzel-like algorithm.

See also evalpoly.

Examples

```
julia> @evalpoly(3, 1, 0, 1)
10

julia> @evalpoly(2, 1, 0, 1)
5

julia> @evalpoly(2, 1, 1, 1)
7
```

[1] Donald Knuth, Art of Computer Programming, Volume 2: Seminumerical Algorithms, Sec. 4.6.4.

source

`Base.FastMath.@fastmath` – Macro.

```
@fastmath expr
```

Execute a transformed version of the expression, which calls functions that may violate strict IEEE semantics. This allows the fastest possible operation, but results are undefined – be careful when doing this, as it may change numerical results.

This sets the LLVM Fast-Math flags, and corresponds to the `-ffast-math` option in clang. See the notes on performance annotations for more details.

Examples

```
julia> @fastmath 1+2
3

julia> @fastmath(sin(3))
0.1411200080598672
```

source

46.3 Customizable binary operators

Some unicode characters can be used to define new binary operators that support infix notation. For example `⊗(x,y) = kron(x,y)` defines the `⊗` (otimes) function to be the Kronecker product, and one can call it as binary operator using infix syntax: `C = A ⊗ B` as well as with the usual prefix syntax `C = ⊗(A,B)`.

Other characters that support such extensions include \odot ⊙ and \oplus ⊕

The complete list is in the parser code: https://github.com/JuliaLang/julia/blob/master/src/julia-parser.scm

Those that are parsed like * (in terms of precedence) include `* / ÷ % & ⋅ ∘ × |\\| ∩ ∧ ⊗ ⊘ ⊙ ⊚ ⊛ ⊠ ⊡ ⊓ ∗ ∙ |⁄ □ □ □ ⋄ ⋆ □ □ □ □ □ ⅄ □ □ □ □ □ □ □ □ □ □ □ □ □ □ □ □ □ □ □ □ □ □ □ □ □ □ □ □ □ □ □ □ □ □ □ □ □ ⊍ ▷ □ □ □ □` and those that are parsed like + include `+ - |\|| ⊕ ⊖ ⊞ ⊟ |++| ∪ ∨ ⊔ ± ∓ □ ∸ ≏ ⊎ ⊻ □ ⋎ □ □ ⧺ ⧻ □ □ □ □ □ □ □ □ □ □ □ □ □ □ □ □ □ □ □ □ □ □ □ □ □ □ □ □ □ □ □ □` There are many others that are related to arrows, comparisons, and powers.

Chapter 47

Numbers

47.1 Standard Numeric Types

A type tree for all subtypes of `Number` in `Base` is shown below. Abstract types have been marked, the rest are concrete types.

```
Number  (Abstract Type)
├─ Complex
└─ Real  (Abstract Type)
   ├─ AbstractFloat  (Abstract Type)
   │  ├─ Float16
   │  ├─ Float32
   │  ├─ Float64
   │  └─ BigFloat
   ├─ Integer  (Abstract Type)
   │  ├─ Bool
   │  ├─ Signed  (Abstract Type)
   │  │  ├─ Int8
   │  │  ├─ Int16
   │  │  ├─ Int32
   │  │  ├─ Int64
   │  │  ├─ Int128
   │  │  └─ BigInt
   │  └─ Unsigned  (Abstract Type)
   │     ├─ UInt8
   │     ├─ UInt16
   │     ├─ UInt32
   │     ├─ UInt64
   │     └─ UInt128
   ├─ Rational
   └─ AbstractIrrational  (Abstract Type)
      └─ Irrational
```

Abstract number types

`Core.Number` – Type.

```
Number
```

Abstract supertype for all number types.

source

`Core.Real` – Type.

```
Real <: Number
```

Abstract supertype for all real numbers.

source

`Core.AbstractFloat` – Type.

```
AbstractFloat <: Real
```

Abstract supertype for all floating point numbers.

source

`Core.Integer` – Type.

```
Integer <: Real
```

Abstract supertype for all integers (e.g. `Signed`, `Unsigned`, and `Bool`).

See also `isinteger`, `trunc`, `div`.

Examples

```
julia> 42 isa Integer
true

julia> 1.0 isa Integer
false

julia> isinteger(1.0)
true
```

source

`Core.Signed` – Type.

```
Signed <: Integer
```

Abstract supertype for all signed integers.

source

`Core.Unsigned` – Type.

```
Unsigned <: Integer
```

Abstract supertype for all unsigned integers.

Built-in unsigned integers are printed in hexadecimal, with prefix `0x`, and can be entered in the same way.

Examples

```
julia> typemax(UInt8)
0xff

julia> Int(0x00d)
13

julia> unsigned(true)
0x0000000000000001
```

source

Base.AbstractIrrational – Type.

```
AbstractIrrational <: Real
```

Number type representing an exact irrational value, which is automatically rounded to the correct precision in arithmetic operations with other numeric quantities.

Subtypes `MyIrrational <: AbstractIrrational` should implement at least `==(::MyIrrational, ::MyIrrational)`, `hash(x::MyIrrational, h::UInt)`, and `convert(::Type{F}, x::MyIrrational) where {F <: Union{BigFloat,Float32,Float64}}`.

If a subtype is used to represent values that may occasionally be rational (e.g. a square-root type that represents √n for integers n will give a rational result when n is a perfect square), then it should also implement `isinteger`, `iszero`, `isone`, and `==` with `Real` values (since all of these default to `false` for `AbstractIrrational` types), as well as defining `hash` to equal that of the corresponding `Rational`.

source

Concrete number types

Core.Float16 – Type.

```
Float16 <: AbstractFloat <: Real
```

16-bit floating point number type (IEEE 754 standard). Binary format is 1 sign, 5 exponent, 10 fraction bits.

source

Core.Float32 – Type.

```
Float32 <: AbstractFloat <: Real
```

32-bit floating point number type (IEEE 754 standard). Binary format is 1 sign, 8 exponent, 23 fraction bits.

The exponent for scientific notation should be entered as lower-case `f`, thus `2f3 === 2.0f0 * 10^3 === Float32(2_000)`. For array literals and comprehensions, the element type can be specified before the square brackets: `Float32[1,4,9] == Float32[i^2 for i in 1:3]`.

See also `Inf32`, `NaN32`, `Float16`, `exponent`, `frexp`.

source

Core.Float64 – Type.

```
Float64 <: AbstractFloat <: Real
```

64-bit floating point number type (IEEE 754 standard). Binary format is 1 sign, 11 exponent, 52 fraction bits. See `bitstring`, `signbit`, `exponent`, `frexp`, and `significand` to access various bits.

This is the default for floating point literals, `1.0 isa Float64`, and for many operations such as `1/2, 2pi, log(2), range(0,90,length=4)`. Unlike integers, this default does not change with `Sys.WORD_SIZE`.

The exponent for scientific notation can be entered as `e` or `E`, thus `2e3 === 2.0E3 === 2.0 * 10^3`. Doing so is strongly preferred over `10^n` because integers overflow, thus `2.0 * 10^19 < 0` but `2e19 > 0`.

See also `Inf`, `NaN`, `floatmax`, `Float32`, `Complex`.

source

`Base.MPFR.BigFloat` – Type.

```
BigFloat <: AbstractFloat
```

Arbitrary precision floating point number type.

source

`Core.Bool` – Type.

```
Bool <: Integer
```

Boolean type, containing the values `true` and `false`.

`Bool` is a kind of number: `false` is numerically equal to `0` and `true` is numerically equal to `1`. Moreover, `false` acts as a multiplicative "strong zero" against `NaN` and `Inf`:

```
julia> [true, false] == [1, 0]
true

julia> 42.0 + true
43.0

julia> 0 .* (NaN, Inf, -Inf)
(NaN, NaN, NaN)

julia> false .* (NaN, Inf, -Inf)
(0.0, 0.0, -0.0)
```

Branches via `if` and other conditionals only accept `Bool`. There are no "truthy" values in Julia.

Comparisons typically return `Bool`, and broadcasted comparisons may return `BitArray` instead of an `Array{Bool}`.

```
julia> [1 2 3 4 5] .< pi
1×5 BitMatrix:
 1  1  1  0  0

julia> map(>(pi), [1 2 3 4 5])
1×5 Matrix{Bool}:
 0  0  0  1  1
```

See also trues, falses, ifelse.

source

Core.Int8 – Type.

```
Int8 <: Signed <: Integer
```

8-bit signed integer type.

Represents numbers n ∈ -128:127. Note that such integers overflow without warning, thus typemax(Int8) + Int8(1) < 0.

See also Int, widen, BigInt.

source

Core.UInt8 – Type.

```
UInt8 <: Unsigned <: Integer
```

8-bit unsigned integer type.

Printed in hexadecimal, thus 0x07 == 7.

source

Core.Int16 – Type.

```
Int16 <: Signed <: Integer
```

16-bit signed integer type.

Represents numbers n ∈ -32768:32767. Note that such integers overflow without warning, thus typemax(Int16) + Int16(1) < 0.

See also Int, widen, BigInt.

source

Core.UInt16 – Type.

```
UInt16 <: Unsigned <: Integer
```

16-bit unsigned integer type.

Printed in hexadecimal, thus 0x000f == 15.

source

Core.Int32 – Type.

```
Int32 <: Signed <: Integer
```

32-bit signed integer type.

Note that such integers overflow without warning, thus typemax(Int32) + Int32(1) < 0.

See also Int, widen, BigInt.

source

`Core.UInt32` – Type.

```
UInt32 <: Unsigned <: Integer
```

32-bit unsigned integer type.

Printed in hexadecimal, thus 0x0000001f == 31.

source

`Core.Int64` – Type.

```
Int64 <: Signed <: Integer
```

64-bit signed integer type.

Note that such integers overflow without warning, thus `typemax(Int64) + Int64(1) < 0`.

See also `Int`, `widen`, `BigInt`.

source

`Core.UInt64` – Type.

```
UInt64 <: Unsigned <: Integer
```

64-bit unsigned integer type.

Printed in hexadecimal, thus 0x000000000000003f == 63.

source

`Core.Int128` – Type.

```
Int128 <: Signed <: Integer
```

128-bit signed integer type.

Note that such integers overflow without warning, thus `typemax(Int128) + Int128(1) < 0`.

See also `Int`, `widen`, `BigInt`.

source

`Core.UInt128` – Type.

```
UInt128 <: Unsigned <: Integer
```

128-bit unsigned integer type.

Printed in hexadecimal, thus 0x0000000000000000000000000000007f == 127.

source

`Core.Int` – Type.

```
Int
```

Sys.WORD_SIZE-bit signed integer type, `Int <: Signed <: Integer <: Real`.

This is the default type of most integer literals and is an alias for either `Int32` or `Int64`, depending on `Sys.WORD_SIZE`. It is the type returned by functions such as `length`, and the standard type for indexing arrays.

Note that integers overflow without warning, thus `typemax(Int) + 1 < 0` and `10^19 < 0`. Overflow can be avoided by using `BigInt`. Very large integer literals will use a wider type, for instance `10_000_000_000_000_000_000 isa Int128`.

Integer division is `div` alias ÷, whereas `/` acting on integers returns `Float64`.

See also `Int64`, `widen`, `typemax`, `bitstring`.

source

`Core.UInt` – Type.

```
UInt
```

Sys.WORD_SIZE-bit unsigned integer type, `UInt <: Unsigned <: Integer`.

Like `Int`, the alias `UInt` may point to either `UInt32` or `UInt64`, according to the value of `Sys.WORD_SIZE` on a given computer.

Printed and parsed in hexadecimal: `UInt(15) === 0x000000000000000f`.

source

`Base.GMP.BigInt` – Type.

```
BigInt <: Signed
```

Arbitrary precision integer type.

source

`Base.Complex` – Type.

```
Complex{T<:Real} <: Number
```

Complex number type with real and imaginary part of type T.

`ComplexF16`, `ComplexF32` and `ComplexF64` are aliases for `Complex{Float16}`, `Complex{Float32}` and `Complex{Float64}` respectively.

See also: `Real`, `complex`, `real`.

source

`Base.Rational` – Type.

```
Rational{T<:Integer} <: Real
```

Rational number type, with numerator and denominator of type T. Rationals are checked for overflow.

source

`Base.Irrational` – Type.

```
Irrational{sym} <: AbstractIrrational
```

Number type representing an exact irrational value denoted by the symbol `sym`, such as π, ℯ and γ.

See also `AbstractIrrational`.

source

47.2 Data Formats

`Base.digits` – Function.

```
digits([T<:Integer], n::Integer; base::T = 10, pad::Integer = 1)
```

Return an array with element type `T` (default `Int`) of the digits of `n` in the given base, optionally padded with zeros to a specified size. More significant digits are at higher indices, such that `n == sum(digits[k]*base^(k-1) for k in 1:length(digits))`.

See also `ndigits`, `digits!`, and for base 2 also `bitstring`, `count_ones`.

Examples

```
julia> digits(10)
2-element Vector{Int64}:
 0
 1

julia> digits(10, base = 2)
4-element Vector{Int64}:
 0
 1
 0
 1

julia> digits(-256, base = 10, pad = 5)
5-element Vector{Int64}:
 -6
 -5
 -2
  0
  0

julia> n = rand(-999:999);

julia> n == evalpoly(13, digits(n, base = 13))
true
```

source

`Base.digits!` – Function.

```
digits!(array, n::Integer; base::Integer = 10)
```

Fills an array of the digits of n in the given base. More significant digits are at higher indices. If the array length is insufficient, the least significant digits are filled up to the array length. If the array length is excessive, the excess portion is filled with zeros.

Examples

```
julia> digits!([2, 2, 2, 2], 10, base = 2)
4-element Vector{Int64}:
 0
 1
 0
 1

julia> digits!([2, 2, 2, 2, 2, 2], 10, base = 2)
6-element Vector{Int64}:
 0
 1
 0
 1
 0
 0
```

source

Base.bitstring – Function.

```
bitstring(n)
```

A string giving the literal bit representation of a primitive type (in bigendian order, i.e. most-significant bit first).

See also count_ones, count_zeros, digits.

Examples

```
julia> bitstring(Int32(4))
"00000000000000000000000000000100"

julia> bitstring(2.2)
"0100000000000001100110011001100110011001100110011001100110011010"
```

source

Base.parse – Function.

```
parse(::Type{SimpleColor}, rgb::String)
```

An analogue of tryparse(SimpleColor, rgb::String) (which see), that raises an error instead of returning nothing.

```
parse(type, str; base)
```

Parse a string as a number. For `Integer` types, a base can be specified (the default is 10). For floating-point types, the string is parsed as a decimal floating-point number. `Complex` types are parsed from decimal strings of the form `"R±Iim"` as a `Complex(R,I)` of the requested type; `"i"` or `"j"` can also be used instead of `"im"`, and `"R"` or `"Iim"` are also permitted. If the string does not contain a valid number, an error is raised.

Julia 1.1

`parse(Bool, str)` requires at least Julia 1.1.

Examples

```
julia> parse(Int, "1234")
1234

julia> parse(Int, "1234", base = 5)
194

julia> parse(Int, "afc", base = 16)
2812

julia> parse(Float64, "1.2e-3")
0.0012

julia> parse(Complex{Float64}, "3.2e-1 + 4.5im")
0.32 + 4.5im
```

source

```
parse(::Type{Platform}, triplet::AbstractString)
```

Parses a string platform triplet back into a `Platform` object.

source

`Base.tryparse` – Function.

```
tryparse(::Type{SimpleColor}, rgb::String)
```

Attempt to parse `rgb` as a `SimpleColor`. If `rgb` starts with `#` and has a length of 7, it is converted into a `RGBTuple`-backed `SimpleColor`. If `rgb` starts with `a-z`, `rgb` is interpreted as a color name and converted to a `Symbol`-backed `SimpleColor`.

Otherwise, `nothing` is returned.

Examples

```
julia> tryparse(SimpleColor, "blue")
SimpleColor(blue)

julia> tryparse(SimpleColor, "#9558b2")
SimpleColor(#9558b2)

julia> tryparse(SimpleColor, "#nocolor")
```

```
tryparse(type, str; base)
```

Like `parse`, but returns either a value of the requested type, or `nothing` if the string does not contain a valid number.

source

`Base.big` – Function.

```
big(x)
```

Convert a number to a maximum precision representation (typically `BigInt` or `BigFloat`). See `BigFloat` for information about some pitfalls with floating-point numbers.

source

`Base.signed` – Function.

```
signed(x)
```

Convert a number to a signed integer. If the argument is unsigned, it is reinterpreted as signed without checking for overflow.

See also: `unsigned`, `sign`, `signbit`.

source

```
signed(T::Integer)
```

Convert an integer bitstype to the signed type of the same size.

Examples

```
julia> signed(UInt16)
Int16
julia> signed(UInt64)
Int64
```

source

`Base.unsigned` – Function.

```
unsigned(T::Integer)
```

Convert an integer bitstype to the unsigned type of the same size.

Examples

```
julia> unsigned(Int16)
UInt16
julia> unsigned(UInt64)
UInt64
```

source

`Base.float` – Method.

```
float(x)
```

Convert a number or array to a floating point data type.

See also: `complex`, `oftype`, `convert`.

Examples

```
julia> float(typemax(Int32))
2.147483647e9
```

source

`Base.Math.significand` – Function.

```
significand(x)
```

Extract the significand (a.k.a. mantissa) of a floating-point number. If x is a non-zero finite number, then the result will be a number of the same type and sign as x, and whose absolute value is on the interval $[1, 2)$. Otherwise x is returned.

See also `frexp`, `exponent`.

Examples

```
julia> significand(15.2)
1.9

julia> significand(-15.2)
-1.9

julia> significand(-15.2) * 2^3
-15.2

julia> significand(-Inf), significand(Inf), significand(NaN)
(-Inf, Inf, NaN)
```

source

`Base.Math.exponent` – Function.

```
exponent(x::Real)::Int
```

Return the largest integer y such that `2^y ≤ abs(x)`. For a normalized floating-point number x, this corresponds to the exponent of x.

Throws a `DomainError` when x is zero, infinite, or `NaN`. For any other non-subnormal floating-point number x, this corresponds to the exponent bits of x.

See also `signbit`, `significand`, `frexp`, `issubnormal`, `log2`, `ldexp`.

Examples

```
julia> exponent(8)
3
```

```
julia> exponent(6.5)
2

julia> exponent(-1//4)
-2

julia> exponent(3.142e-4)
-12

julia> exponent(floatmin(Float32)), exponent(nextfloat(0.0f0))
(-126, -149)

julia> exponent(0.0)
ERROR: DomainError with 0.0:
Cannot be ±0.0.
[...]
```

source

Base.complex – Method.

```
complex(r, [i])
```

Convert real numbers or arrays to complex. `i` defaults to zero.

Examples

```
julia> complex(7)
7 + 0im
```

source

Base.bswap – Function.

```
bswap(n)
```

Reverse the byte order of `n`.

(See also `ntoh` and `hton` to convert between the current native byte order and big-endian order.)

Examples

```
julia> a = bswap(0x10203040)
0x40302010

julia> bswap(a)
0x10203040

julia> string(1, base = 2)
"1"

julia> string(bswap(1), base = 2)
"10000000000000000000000000000000000000000000000000000000000000000"
```

source

Base.hex2bytes – Function.

```
hex2bytes(itr)
```

Given an iterable `itr` of ASCII codes for a sequence of hexadecimal digits, returns a `Vector{UInt8}` of bytes corresponding to the binary representation: each successive pair of hexadecimal digits in `itr` gives the value of one byte in the return vector.

The length of `itr` must be even, and the returned array has half of the length of `itr`. See also `hex2bytes!` for an in-place version, and `bytes2hex` for the inverse.

Julia 1.7

Calling hex2bytes with iterators producing `UInt8` values requires Julia 1.7 or later. In earlier versions, you can `collect` the iterator before calling hex2bytes.

Examples

```
julia> s = string(12345, base = 16)
"3039"

julia> hex2bytes(s)
2-element Vector{UInt8}:
 0x30
 0x39

julia> a = b"01abEF"
6-element Base.CodeUnits{UInt8, String}:
 0x30
 0x31
 0x61
 0x62
 0x45
 0x46

julia> hex2bytes(a)
3-element Vector{UInt8}:
 0x01
 0xab
 0xef
```

source

Base.hex2bytes! – Function.

```
hex2bytes!(dest::AbstractVector{UInt8}, itr)
```

Convert an iterable `itr` of bytes representing a hexadecimal string to its binary representation, similar to `hex2bytes` except that the output is written in-place to `dest`. The length of `dest` must be half the length of `itr`.

Julia 1.7

Calling hex2bytes! with iterators producing UInt8 requires version 1.7. In earlier versions, you can collect the iterable before calling instead.

source

Base.bytes2hex – Function.

```
bytes2hex(itr)::String
bytes2hex(io::IO, itr)::Nothing
```

Convert an iterator `itr` of bytes to its hexadecimal string representation, either returning a `String` via `bytes2hex(itr)` or writing the string to an `io` stream via `bytes2hex(io, itr)`. The hexadecimal characters are all lowercase.

Julia 1.7

Calling bytes2hex with arbitrary iterators producing UInt8 values requires Julia 1.7 or later. In earlier versions, you can `collect` the iterator before calling bytes2hex.

Examples

```
julia> a = string(12345, base = 16)
"3039"

julia> b = hex2bytes(a)
2-element Vector{UInt8}:
 0x30
 0x39

julia> bytes2hex(b)
"3039"
```

source

47.3 General Number Functions and Constants

Base.one – Function.

```
one(x)
one(T::Type)
```

Return a multiplicative identity for x: a value such that `one(x)*x == x*one(x) == x`. If the multiplicative identity can be deduced from the type alone, then a type may be given as an argument to one (e.g. `one(Int)` will work because the multiplicative identity is the same for all instances of `Int`, but `one(Matrix{Int})` is not defined because matrices of different shapes have different multiplicative identities.)

If possible, `one(x)` returns a value of the same type as x, and `one(T)` returns a value of type T. However, this may not be the case for types representing dimensionful quantities (e.g. time in days), since the multiplicative identity must be dimensionless. In that case, `one(x)` should return an identity value of the same precision (and shape, for matrices) as x.

If you want a quantity that is of the same type as x, or of type T, even if x is dimensionful, use `oneunit` instead.

See also the `identity` function, and `I` in `LinearAlgebra` for the identity matrix.

Examples

```
julia> one(3.7)
1.0

julia> one(Int)
1

julia> import Dates; one(Dates.Day(1))
1
```

source

`Base.oneunit` – Function.

```
oneunit(x::T)
oneunit(T::Type)
```

Return `T(one(x))`, where T is either the type of the argument, or the argument itself in cases where the `oneunit` can be deduced from the type alone. This differs from `one` for dimensionful quantities: one is dimensionless (a multiplicative identity) while `oneunit` is dimensionful (of the same type as x, or of type T).

Examples

```
julia> oneunit(3.7)
1.0

julia> import Dates; oneunit(Dates.Day)
1 day
```

source

`Base.zero` – Function.

```
zero(x)
zero(::Type)
```

Get the additive identity element for `x`. If the additive identity can be deduced from the type alone, then a type may be given as an argument to `zero`.

For example, `zero(Int)` will work because the additive identity is the same for all instances of `Int`, but `zero(Vector{Int})` is not defined because vectors of different lengths have different additive identities.

See also `iszero`, `one`, `oneunit`, `oftype`.

Examples

```
julia> zero(1)
0

julia> zero(big"2.0")
0.0

julia> zero(rand(2,2))
2×2 Matrix{Float64}:
 0.0  0.0
 0.0  0.0
```

source

Base.im – Constant.

```
im
```

The imaginary unit.

See also: imag, angle, complex.

Examples

```
julia> im * im
-1 + 0im

julia> (2.0 + 3im)^2
-5.0 + 12.0im
```

source

Base.MathConstants.pi – Constant.

```
π
pi
```

The constant pi.

Unicode π can be typed by writing \pi then pressing tab in the Julia REPL, and in many editors.

See also: sinpi, sincospi, deg2rad.

Examples

```
julia> pi
π = 3.1415926535897...

julia> 1/2pi
0.15915494309189535
```

source

Base.MathConstants.□ – Constant.

```
□
e
```

The constant *e*.

Unicode □ can be typed by writing \euler and pressing tab in the Julia REPL, and in many editors.

See also: exp, cis, cispi.

Examples

```
julia> □
□ = 2.7182818284590...

julia> log(□)
1

julia> □^(im)π ≈ -1
true
```

source

Base.MathConstants.catalan – Constant.

```
catalan
```

Catalan's constant.

Examples

```
julia> Base.MathConstants.catalan
catalan = 0.9159655941772...

julia> sum(log(x)/(1+x^2) for x in 1:0.01:10^6) * 0.01
0.9159466120554123
```

source

Base.MathConstants.eulergamma – Constant.

```
γ
eulergamma
```

Euler's constant.

Examples

```
julia> Base.MathConstants.eulergamma
γ = 0.5772156649015...

julia> dx = 10^-6;

julia> sum(-exp(-x) * log(x) for x in dx:dx:100) * dx
0.5772078382499133
```

source

Base.MathConstants.golden – Constant.

```
φ
golden
```

The golden ratio.

Examples

```
julia> Base.MathConstants.golden
φ = 1.6180339887498...

julia> (2ans - 1)^2 ≈ 5
true
```

source

Base.Inf – Constant.

```
Inf, Inf64
```

Positive infinity of type Float64.

See also: isfinite, typemax, NaN, Inf32.

Examples

```
julia> π/0
Inf

julia> +1.0 / -0.0
-Inf

julia> □^-Inf
0.0
```

source

Base.Inf64 – Constant.

```
Inf, Inf64
```

Positive infinity of type Float64.

See also: isfinite, typemax, NaN, Inf32.

Examples

```
julia> π/0
Inf

julia> +1.0 / -0.0
-Inf

julia> □^-Inf
0.0
```

source

Base.Inf32 – Constant.

```
Inf32
```

Positive infinity of type Float32.

source

Base.Inf16 – Constant.

```
Inf16
```

Positive infinity of type Float16.

source

Base.NaN – Constant.

```
NaN, NaN64
```

A not-a-number value of type Float64.

See also: isnan, missing, NaN32, Inf.

Examples

```
julia> 0/0
NaN

julia> Inf - Inf
NaN

julia> NaN == NaN, isequal(NaN, NaN), isnan(NaN)
(false, true, true)
```

Note

Always use isnan or isequal for checking for NaN. Using x === NaN may give unexpected results:

```
julia> reinterpret(UInt32, NaN32)
0x7fc00000

julia> NaN32p1 = reinterpret(Float32, 0x7fc00001)
NaN32

julia> NaN32p1 === NaN32, isequal(NaN32p1, NaN32), isnan(NaN32p1)
(false, true, true)
```

source

Base.NaN64 – Constant.

```
NaN, NaN64
```

A not-a-number value of type `Float64`.

See also: `isnan`, `missing`, `NaN32`, `Inf`.

Examples

```
julia> 0/0
NaN

julia> Inf - Inf
NaN

julia> NaN == NaN, isequal(NaN, NaN), isnan(NaN)
(false, true, true)
```

Note

Always use `isnan` or `isequal` for checking for NaN. Using `x === NaN` may give unexpected results:

```
julia> reinterpret(UInt32, NaN32)
0x7fc00000

julia> NaN32p1 = reinterpret(Float32, 0x7fc00001)
NaN32

julia> NaN32p1 === NaN32, isequal(NaN32p1, NaN32), isnan(NaN32p1)
(false, true, true)
```

source

`Base.NaN32` – Constant.

```
NaN32
```

A not-a-number value of type `Float32`.

See also: `NaN`.

source

`Base.NaN16` – Constant.

```
NaN16
```

A not-a-number value of type `Float16`.

See also: `NaN`.

source

`Base.issubnormal` – Function.

```
issubnormal(f)::Bool
```

Test whether a floating point number is subnormal.

An IEEE floating point number is subnormal when its exponent bits are zero and its significand is not zero.

Examples

```
julia> floatmin(Float32)
1.1754944f-38

julia> issubnormal(1.0f-37)
false

julia> issubnormal(1.0f-38)
true
```

source

Base.isfinite – Function.

```
isfinite(f)::Bool
```

Test whether a number is finite.

Examples

```
julia> isfinite(5)
true

julia> isfinite(NaN32)
false
```

source

Base.isinf – Function.

```
isinf(f)::Bool
```

Test whether a number is infinite.

See also: Inf, iszero, isfinite, isnan.

source

Base.isnan – Function.

```
isnan(f)::Bool
```

Test whether a number value is a NaN, an indeterminate value which is neither an infinity nor a finite number ("not a number").

See also: iszero, isone, isinf, ismissing.

source

Base.iszero – Function.

```
iszero(x)
```

Return `true` if `x == zero(x)`; if x is an array, this checks whether all of the elements of x are zero.

See also: `isone`, `isinteger`, `isfinite`, `isnan`.

Examples

```
julia> iszero(0.0)
true

julia> iszero([1, 9, 0])
false

julia> iszero([false, 0, 0])
true
```

source

`Base.isone` – Function.

```
isone(x)
```

Return `true` if `x == one(x)`; if x is an array, this checks whether x is an identity matrix.

Examples

```
julia> isone(1.0)
true

julia> isone([1 0; 0 2])
false

julia> isone([1 0; 0 true])
true
```

source

`Base.nextfloat` – Function.

```
nextfloat(x::AbstractFloat)
```

Return the smallest floating point number y of the same type as x such that `x < y`. If no such y exists (e.g. if x is `Inf` or `NaN`), then return x.

See also: `prevfloat`, `eps`, `issubnormal`.

source

```
nextfloat(x::AbstractFloat, n::Integer)
```

The result of n iterative applications of `nextfloat` to x if `n >= 0`, or `-n` applications of `prevfloat` if `n < 0`.

source

`Base.prevfloat` – Function.

```
prevfloat(x::AbstractFloat)
```

Return the largest floating point number `y` of the same type as `x` such that `y < x`. If no such `y` exists (e.g. if `x` is `-Inf` or `NaN`), then return `x`.

source

```
prevfloat(x::AbstractFloat, n::Integer)
```

The result of `n` iterative applications of `prevfloat` to `x` if `n >= 0`, or `-n` applications of `nextfloat` if `n < 0`.

source

`Base.isinteger` – Function.

```
isinteger(x)::Bool
```

Test whether `x` is numerically equal to some integer.

Examples

```
julia> isinteger(4.0)
true
```

source

`Base.isreal` – Function.

```
isreal(x)::Bool
```

Test whether `x` or all its elements are numerically equal to some real number including infinities and NaNs. `isreal(x)` is true if `isequal(x, real(x))` is true.

Examples

```
julia> isreal(5.)
true

julia> isreal(1 - 3im)
false

julia> isreal(Inf + 0im)
true

julia> isreal([4.; complex(0,1)])
false
```

source

`Core.Float32` – Method.

```
Float32(x [, mode::RoundingMode])
```

Create a `Float32` from `x`. If `x` is not exactly representable then `mode` determines how `x` is rounded.

Examples

```
julia> Float32(1/3, RoundDown)
0.3333333f0

julia> Float32(1/3, RoundUp)
0.33333334f0
```

See `RoundingMode` for available rounding modes.

source

`Core.Float64` – Method.

```
Float64(x [, mode::RoundingMode])
```

Create a `Float64` from `x`. If `x` is not exactly representable then `mode` determines how `x` is rounded.

Examples

```
julia> Float64(pi, RoundDown)
3.141592653589793

julia> Float64(pi, RoundUp)
3.1415926535897936
```

See `RoundingMode` for available rounding modes.

source

`Base.Rounding.rounding` – Function.

```
rounding(T)
```

Get the current floating point rounding mode for type `T`, controlling the rounding of basic arithmetic functions (`+`, `-`, `*`, `/` and `sqrt`) and type conversion.

See `RoundingMode` for available modes.

source

`Base.Rounding.setrounding` – Method.

```
setrounding(T, mode)
```

Set the rounding mode of floating point type `T`, controlling the rounding of basic arithmetic functions (`+`, `-`, `*`, `/` and `sqrt`) and type conversion. Other numerical functions may give incorrect or invalid values when using rounding modes other than the default `RoundNearest`.

Note that this is currently only supported for `T == BigFloat`.

Warning

This function is not thread-safe. It will affect code running on all threads, but its behavior is undefined if called concurrently with computations that use the setting.

source

Base.Rounding.setrounding – Method.

```
setrounding(f::Function, T, mode)
```

Change the rounding mode of floating point type T for the duration of `f`. It is logically equivalent to:

```
old = rounding(T)
setrounding(T, mode)
f()
setrounding(T, old)
```

See RoundingMode for available rounding modes.

source

Base.Rounding.get_zero_subnormals – Function.

```
get_zero_subnormals()::Bool
```

Return `false` if operations on subnormal floating-point values ("denormals") obey rules for IEEE arithmetic, and `true` if they might be converted to zeros.

Warning

This function only affects the current thread.

source

Base.Rounding.set_zero_subnormals – Function.

```
set_zero_subnormals(yes::Bool)::Bool
```

If `yes` is `false`, subsequent floating-point operations follow rules for IEEE arithmetic on subnormal values ("denormals"). Otherwise, floating-point operations are permitted (but not required) to convert subnormal inputs or outputs to zero. Returns `true` unless `yes==true` but the hardware does not support zeroing of subnormal numbers.

`set_zero_subnormals(true)` can speed up some computations on some hardware. However, it can break identities such as `(x-y==0) == (x==y)`.

Warning

This function only affects the current thread.

source

Integers

Base.count_ones – Function.

```
count_ones(x::Integer)::Integer
```

Number of ones in the binary representation of `x`.

Examples

```
julia> count_ones(7)
3

julia> count_ones(Int32(-1))
32
```

source

Base.count_zeros – Function.

```
count_zeros(x::Integer)::Integer
```

Number of zeros in the binary representation of x.

Examples

```
julia> count_zeros(Int32(2 ^ 16 - 1))
16

julia> count_zeros(-1)
0
```

source

Base.leading_zeros – Function.

```
leading_zeros(x::Integer)::Integer
```

Number of zeros leading the binary representation of x.

Examples

```
julia> leading_zeros(Int32(1))
31
```

source

Base.leading_ones – Function.

```
leading_ones(x::Integer)::Integer
```

Number of ones leading the binary representation of x.

Examples

```
julia> leading_ones(UInt32(2 ^ 32 - 2))
31
```

source

Base.trailing_zeros – Function.

```
trailing_zeros(x::Integer)::Integer
```

Number of zeros trailing the binary representation of x.

Examples

```
julia> trailing_zeros(2)
1
```

source

Base.trailing_ones – Function.

```
trailing_ones(x::Integer)::Integer
```

Number of ones trailing the binary representation of x.

Examples

```
julia> trailing_ones(3)
2
```

source

Base.isodd – Function.

```
isodd(x::Number)::Bool
```

Return `true` if x is an odd integer (that is, an integer not divisible by 2), and `false` otherwise.

Julia 1.7

Non-`Integer` arguments require Julia 1.7 or later.

Examples

```
julia> isodd(9)
true

julia> isodd(10)
false
```

source

Base.iseven – Function.

```
iseven(x::Number)::Bool
```

Return `true` if x is an even integer (that is, an integer divisible by 2), and `false` otherwise.

Julia 1.7

Non-`Integer` arguments require Julia 1.7 or later.

Examples

```
julia> iseven(9)
false

julia> iseven(10)
true
```

source

Core.@int128_str – Macro.

```
@int128_str str
```

Parse `str` as an `Int128`. Throw an `ArgumentError` if the string is not a valid integer.

Examples

```
julia> int128"123456789123"
123456789123

julia> int128"123456789123.4"
ERROR: LoadError: ArgumentError: invalid base 10 digit '.' in "123456789123.4"
[...]
```

source

Core.@uint128_str – Macro.

```
@uint128_str str
```

Parse `str` as an `UInt128`. Throw an `ArgumentError` if the string is not a valid integer.

Examples

```
julia> uint128"123456789123"
0x00000000000000000000001cbe991a83

julia> uint128"-123456789123"
ERROR: LoadError: ArgumentError: invalid base 10 digit '-' in "-123456789123"
[...]
```

source

47.4 BigFloats and BigInts

The `BigFloat` and `BigInt` types implement arbitrary-precision floating point and integer arithmetic, respectively. For `BigFloat` the GNU MPFR library is used, and for `BigInt` the [GNU Multiple Precision Arithmetic Library (GMP)] (https://gmplib.org) is used.

Base.MPFR.BigFloat – Method.

```
BigFloat(x::Union{Real, AbstractString} [, rounding::RoundingMode=rounding(BigFloat)];
↪  [precision::Integer=precision(BigFloat)])
```

Create an arbitrary precision floating point number from `x`, with precision `precision`. The `rounding` argument specifies the direction in which the result should be rounded if the conversion cannot be done exactly. If not provided, these are set by the current global values.

`BigFloat(x::Real)` is the same as `convert(BigFloat,x)`, except if `x` itself is already `BigFloat`, in which case it will return a value with the precision set to the current global precision; `convert` will always return `x`.

`BigFloat(x::AbstractString)` is identical to `parse`. This is provided for convenience since decimal literals are converted to `Float64` when parsed, so `BigFloat(2.1)` may not yield what you expect.

See also:

- `@big_str`
- `rounding` and `setrounding`
- `precision` and `setprecision`

Julia 1.1

`precision` as a keyword argument requires at least Julia 1.1. In Julia 1.0 `precision` is the second positional argument (`BigFloat(x, precision)`).

Examples

```
julia> BigFloat(2.1) # 2.1 here is a Float64
2.100000000000000088817841970012523233890533447265625

julia> BigFloat("2.1") # the closest BigFloat to 2.1
2.099999999999999999999999999999999999999999999999999999999999999999999999999999986

julia> BigFloat("2.1", RoundUp)
2.100000000000000000000000000000000000000000000000000000000000000000000000000000021

julia> BigFloat("2.1", RoundUp, precision=128)
2.100000000000000000000000000000000000007
```

source

`Base.precision` – Function.

```
precision(num::AbstractFloat; base::Integer=2)
precision(T::Type; base::Integer=2)
```

Get the precision of a floating point number, as defined by the effective number of bits in the significand, or the precision of a floating-point type `T` (its current default, if `T` is a variable-precision type like `BigFloat`).

If `base` is specified, then it returns the maximum corresponding number of significand digits in that base.

Julia 1.8

The `base` keyword requires at least Julia 1.8.

source

`Base.MPFR.setprecision` – Function.

```
setprecision(f::Function, [T=BigFloat,] precision::Integer; base=2)
```

Change the T arithmetic precision (in the given base) for the duration of `f`. It is logically equivalent to:

```
old = precision(BigFloat)
setprecision(BigFloat, precision)
f()
setprecision(BigFloat, old)
```

Often used as `setprecision(T, precision) do ... end`

Note: `nextfloat()`, `prevfloat()` do not use the precision mentioned by `setprecision`.

Warning

There is a fallback implementation of this method that calls `precision` and `setprecision`, but it should no longer be relied on. Instead, you should define the 3-argument form directly in a way that uses `ScopedValue`, or recommend that callers use `ScopedValue` and `@with` themselves.

Julia 1.8

The base keyword requires at least Julia 1.8.

source

```
setprecision([T=BigFloat,] precision::Int; base=2)
```

Set the precision (in bits, by default) to be used for T arithmetic. If base is specified, then the precision is the minimum required to give at least `precision` digits in the given base.

Warning

This function is not thread-safe. It will affect code running on all threads, but its behavior is undefined if called concurrently with computations that use the setting.

Julia 1.8

The base keyword requires at least Julia 1.8.

source

`Base.GMP.BigInt` – Method.

```
BigInt(x)
```

Create an arbitrary precision integer. `x` may be an `Int` (or anything that can be converted to an `Int`). The usual mathematical operators are defined for this type, and results are promoted to a `BigInt`.

Instances can be constructed from strings via `parse`, or using the `big` string literal.

Examples

```
julia> parse(BigInt, "42")
42

julia> big"313"
313

julia> BigInt(10)^19
10000000000000000000
```

source

`Core.@big_str` – Macro.

```
@big_str str
```

Parse a string into a `BigInt` or `BigFloat`, and throw an `ArgumentError` if the string is not a valid number. For integers _ is allowed in the string as a separator.

Examples

```
julia> big"123_456"
123456

julia> big"7891.5"
7891.5

julia> big"_"
ERROR: ArgumentError: invalid number format _ for BigInt or BigFloat
[...]
```

Warning

Using `@big_str` for constructing `BigFloat` values may not result in the behavior that might be naively expected: as a macro, `@big_str` obeys the global precision (`setprecision`) and rounding mode (`setrounding`) settings as they are at *load time*. Thus, a function like `() -> precision(big"0.3")` returns a constant whose value depends on the value of the precision at the point when the function is defined, **not** at the precision at the time when the function is called.

source

Chapter 48

Strings

`Core.AbstractString` – Type.

The `AbstractString` type is the supertype of all string implementations in Julia. Strings are encodings of sequences of Unicode code points as represented by the `AbstractChar` type. Julia makes a few assumptions about strings:

- Strings are encoded in terms of fixed-size "code units"
  - Code units can be extracted with `codeunit(s, i)`
  - The first code unit has index `1`
  - The last code unit has index `ncodeunits(s)`
  - Any index `i` such that `1 ≤ i ≤ ncodeunits(s)` is in bounds
- String indexing is done in terms of these code units:
  - Characters are extracted by `s[i]` with a valid string index `i`
  - Each `AbstractChar` in a string is encoded by one or more code units
  - Only the index of the first code unit of an `AbstractChar` is a valid index
  - The encoding of an `AbstractChar` is independent of what precedes or follows it
  - String encodings are self-synchronizing – i.e. `isvalid(s, i)` is O(1)

Some string functions that extract code units, characters or substrings from strings error if you pass them out-of-bounds or invalid string indices. This includes `codeunit(s, i)` and `s[i]`. Functions that do string index arithmetic take a more relaxed approach to indexing and give you the closest valid string index when in-bounds, or when out-of-bounds, behave as if there were an infinite number of characters padding each side of the string. Usually these imaginary padding characters have code unit length 1 but string types may choose different "imaginary" character sizes as makes sense for their implementations (e.g. substrings may pass index arithmetic through to the underlying string they provide a view into). Relaxed indexing functions include those intended for index arithmetic: `thisind`, `nextind` and `prevind`. This model allows index arithmetic to work with out-of-bounds indices as intermediate values so long as one never uses them to retrieve a character, which often helps avoid needing to code around edge cases.

See also `codeunit`, `ncodeunits`, `thisind`, `nextind`, `prevind`.

source

`Core.AbstractChar` – Type.

The `AbstractChar` type is the supertype of all character implementations in Julia. A character represents a Unicode code point, and can be converted to an integer via the `codepoint` function in order to obtain the numerical value of the code point, or constructed from the same integer. These numerical values determine how characters are compared with < and ==, for example. New `T <: AbstractChar` types should define a `codepoint(::T)` method and a `T(::UInt32)` constructor, at minimum.

A given `AbstractChar` subtype may be capable of representing only a subset of Unicode, in which case conversion from an unsupported `UInt32` value may throw an error. Conversely, the built-in `Char` type represents a *superset* of Unicode (in order to losslessly encode invalid byte streams), in which case conversion of a non-Unicode value *to* `UInt32` throws an error. The `isvalid` function can be used to check which codepoints are representable in a given `AbstractChar` type.

Internally, an `AbstractChar` type may use a variety of encodings. Conversion via `codepoint(char)` will not reveal this encoding because it always returns the Unicode value of the character. `print(io, c)` of any `c::AbstractChar` produces an encoding determined by `io` (UTF-8 for all built-in `IO` types), via conversion to `Char` if necessary.

`write(io, c)`, in contrast, may emit an encoding depending on `typeof(c)`, and `read(io, typeof(c))` should read the same encoding as `write`. New `AbstractChar` types must provide their own implementations of `write` and `read`.

source

`Core.Char` – Type.

```
Char(c::Union{Number,AbstractChar})
```

`Char` is a 32-bit `AbstractChar` type that is the default representation of characters in Julia. `Char` is the type used for character literals like `'x'` and it is also the element type of `String`.

In order to losslessly represent arbitrary byte streams stored in a `String`, a `Char` value may store information that cannot be converted to a Unicode codepoint — converting such a `Char` to `UInt32` will throw an error. The `isvalid(c::Char)` function can be used to query whether `c` represents a valid Unicode character.

source

`Base.codepoint` – Function.

```
codepoint(c::AbstractChar)::Integer
```

Return the Unicode codepoint (an unsigned integer) corresponding to the character `c` (or throw an exception if `c` does not represent a valid character). For `Char`, this is a `UInt32` value, but `AbstractChar` types that represent only a subset of Unicode may return a different-sized integer (e.g. `UInt8`).

source

`Base.length` – Method.

```
length(s::AbstractString)::Int
length(s::AbstractString, i::Integer, j::Integer)::Int
```

Return the number of characters in string `s` from indices `i` through `j`.

This is computed as the number of code unit indices from `i` to `j` which are valid character indices. With only a single string argument, this computes the number of characters in the entire string. With `i` and `j` arguments it computes the number of indices between `i` and `j` inclusive that are valid indices in the string `s`. In addition to in-bounds values, `i` may take the out-of-bounds value `ncodeunits(s) + 1` and `j` may take the out-of-bounds value `0`.

Note

The time complexity of this operation is linear in general. That is, it will take the time proportional to the number of bytes or characters in the string because it counts the value on the fly. This is in contrast to the method for arrays, which is a constant-time operation.

See also `isvalid`, `ncodeunits`, `lastindex`, `thisind`, `nextind`, `prevind`.

Examples

```
julia> length("jμΛIα")
5
```

source

`Base.sizeof` – Method.

```
sizeof(str::AbstractString)
```

Size, in bytes, of the string `str`. Equal to the number of code units in `str` multiplied by the size, in bytes, of one code unit in `str`.

Examples

```
julia> sizeof("")
0

julia> sizeof("∀")
3
```

source

`Base.:*` – Method.

```
*(s::Union{AbstractString, AbstractChar}, t::Union{AbstractString,
↪  AbstractChar}...)::AbstractString
```

Concatenate strings and/or characters, producing a `String` or `AnnotatedString` (as appropriate). This is equivalent to calling the `string` or `annotatedstring` function on the arguments. Concatenation of built-in string types always produces a value of type `String` but other string types may choose to return a string of a different type as appropriate.

Examples

```
julia> "Hello " * "world"
"Hello world"

julia> 'j' * "ulia"
"julia"
```

source

Base.:^ – Method.

```
^(s::Union{AbstractString,AbstractChar}, n::Integer)::AbstractString
```

Repeat a string or character n times. This can also be written as `repeat(s, n)`.

See also repeat.

Examples

```
julia> "Test "^3
"Test Test Test "
```

source

Base.string – Function.

```
string(xs...)
```

Create a string from any values using the print function.

`string` should usually not be defined directly. Instead, define a method `print(io::IO, x::MyType)`. If `string(x)` for a certain type needs to be highly efficient, then it may make sense to add a method to `string` and define `print(io::IO, x::MyType) = print(io, string(x))` to ensure the functions are consistent.

See also: String, repr, sprint, show.

Examples

```
julia> string("a", 1, true)
"a1true"
```

source

```
string(n::Integer; base::Integer = 10, pad::Integer = 1)
```

Convert an integer n to a string in the given base, optionally specifying a number of digits to pad to.

See also digits, bitstring, count_zeros, and the Printf standard library.

Examples

```
julia> string(5, base = 13, pad = 4)
"0005"

julia> string(-13, base = 5, pad = 4)
"-0023"

julia> using Printf

julia> @sprintf("%04i", 5)
"0005"

julia> @sprintf("%4i", 5)
"   5"
```

source

Base.repeat – Method.

```
repeat(s::AbstractString, r::Integer)
```

Repeat a string r times. This can be written as s^r.

See also ^.

Examples

```
julia> repeat("ha", 3)
"hahaha"
```

source

Base.repeat – Method.

```
repeat(c::AbstractChar, r::Integer)::String
```

Repeat a character r times. This can equivalently be accomplished by calling c^r.

Examples

```
julia> repeat('A', 3)
"AAA"
```

source

Base.repr – Method.

```
repr(x; context=nothing)
```

Create a string representation of any value using the 2-argument show(io, x) function, which aims to produce a string that is parseable Julia code, where possible. i.e. eval(Meta.parse(repr(x))) == x should hold true. You should not add methods to repr; define a show method instead.

The optional keyword argument context can be set to a :key=>value pair, a tuple of :key=>value pairs, or an IO or IOContext object whose attributes are used for the I/O stream passed to show.

Note that repr(x) is usually similar to how the value of x would be entered in Julia. See also repr(MIME("text/plain"), x) to instead return a "pretty-printed" version of x designed more for human consumption, equivalent to the REPL display of x, using the 3-argument show(io, mime, x).

Julia 1.7

Passing a tuple to keyword context requires Julia 1.7 or later.

Examples

```
julia> repr(1)
"1"

julia> repr(zeros(3))
```

```
"[0.0, 0.0, 0.0]"

julia> repr(big(1/3))
"0.33333333333333331482961625624739099293947219848632812 5"

julia> repr(big(1/3), context=:compact => true)
"0.333333"
```

source

Core.String – Method.

```
String(s::AbstractString)
```

Create a new String from an existing AbstractString.

source

Base.SubString – Type.

```
SubString(s::AbstractString, i::Integer, j::Integer=lastindex(s))
SubString(s::AbstractString, r::UnitRange{<:Integer})
```

Like getindex, but returns a view into the parent string s within range i:j or r respectively instead of making a copy.

The @views macro converts any string slices s[i:j] into substrings SubString(s, i, j) in a block of code.

Examples

```
julia> SubString("abc", 1, 2)
"ab"

julia> SubString("abc", 1:2)
"ab"

julia> SubString("abc", 2)
"bc"
```

source

Base.LazyString – Type.

```
LazyString <: AbstractString
```

A lazy representation of string interpolation. This is useful when a string needs to be constructed in a context where performing the actual interpolation and string construction is unnecessary or undesirable (e.g. in error paths of functions).

This type is designed to be cheap to construct at runtime, trying to offload as much work as possible to either the macro or later printing operations.

Examples

```
julia> n = 5; str = LazyString("n is ", n)
"n is 5"
```

See also `@lazy_str`.

Julia 1.8

`LazyString` requires Julia 1.8 or later.

Extended help

Safety properties for concurrent programs

A lazy string itself does not introduce any concurrency problems even if it is printed in multiple Julia tasks. However, if `print` methods on a captured value can have a concurrency issue when invoked without synchronizations, printing the lazy string may cause an issue. Furthermore, the `print` methods on the captured values may be invoked multiple times, though only exactly one result will be returned.

Julia 1.9

`LazyString` is safe in the above sense in Julia 1.9 and later.

source

`Base.@lazy_str` – Macro.

```
lazy"str"
```

Create a `LazyString` using regular string interpolation syntax. Note that interpolations are *evaluated* at LazyString construction time, but *printing* is delayed until the first access to the string.

See `LazyString` documentation for the safety properties for concurrent programs.

Examples

```
julia> n = 5; str = lazy"n is $n"
"n is 5"

julia> typeof(str)
LazyString
```

Julia 1.8

`lazy"str"` requires Julia 1.8 or later.

source

`Base.transcode` – Function.

```
transcode(T, src)
```

Convert string data between Unicode encodings. `src` is either a `String` or a `Vector{UIntXX}` of UTF-XX code units, where XX is 8, 16, or 32. `T` indicates the encoding of the return value: `String` to return a (UTF-8 encoded) `String` or `UIntXX` to return a `Vector{UIntXX}` of UTF-XX data. (The alias `Cwchar_t` can also be used as the integer type, for converting `wchar_t*` strings used by external C libraries.)

The `transcode` function succeeds as long as the input data can be reasonably represented in the target encoding; it always succeeds for conversions between UTF-XX encodings, even for invalid Unicode data.

Only conversion to/from UTF-8 is currently supported.

Examples

```
julia> str = "αβγ"
"αβγ"

julia> transcode(UInt16, str)
3-element Vector{UInt16}:
 0x03b1
 0x03b2
 0x03b3

julia> transcode(String, transcode(UInt16, str))
"αβγ"
```

source

`Base.unsafe_string` – Function.

```
unsafe_string(p::Ptr{T}, [length::Integer]) where {T<:Union{UInt16,UInt32,Cwchar_t}}
unsafe_string(p::Cwstring)
```

Transcode a string from the address of a C-style (NUL-terminated) string encoded as UTF-16 (`T=UInt16`), UTF-32 (`T=UInt32`), or the system-dependent `wchar_t` (`T=Cwchar_t` or `Cwstring`), returning a `String` (UTF-8 encoding), similar to `transcode` but reading directly from a pointer. (The pointer can be safely freed afterwards.) If `length` is specified (the length of the data in encoding units), the string does not have to be NUL-terminated.

This function is labeled "unsafe" because it will crash if p is not a valid memory address to data of the requested length (or NUL-terminated data).

source

```
unsafe_string(p::Ptr{UInt8}, [length::Integer])
unsafe_string(p::Cstring)
```

Copy a string from the address of a C-style (NUL-terminated) string encoded as UTF-8. (The pointer can be safely freed afterwards.) If `length` is specified (the length of the data in bytes), the string does not have to be NUL-terminated.

This function is labeled "unsafe" because it will crash if p is not a valid memory address to data of the requested length.

source

`Base.ncodeunits` – Method.

```
ncodeunits(s::AbstractString)::Int
```

Return the number of code units in a string. Indices that are in bounds to access this string must satisfy `1 ≤ i ≤ ncodeunits(s)`. Not all such indices are valid – they may not be the start of a character, but they will return a code unit value when calling `codeunit(s,i)`.

Examples

```
julia> ncodeunits("The Julia Language")
18

julia> ncodeunits("∫eˣ")
6

julia> ncodeunits('∫'), ncodeunits('e'), ncodeunits('ˣ')
(3, 1, 2)
```

See also `codeunit`, `checkbounds`, `sizeof`, `length`, `lastindex`.

source

`Base.codeunit` – Function.

```
codeunit(s::AbstractString, i::Integer)::Union{UInt8, UInt16, UInt32}
```

Return the code unit value in the string `s` at index `i`. Note that

```
codeunit(s, i) :: codeunit(s)
```

I.e. the value returned by `codeunit(s, i)` is of the type returned by `codeunit(s)`.

Examples

```
julia> a = codeunit("Hello", 2)
0x65

julia> typeof(a)
UInt8
```

See also `ncodeunits`, `checkbounds`.

source

```
codeunit(s::AbstractString)::Type{<:Union{UInt8, UInt16, UInt32}}
```

Return the code unit type of the given string object. For ASCII, Latin-1, or UTF-8 encoded strings, this would be `UInt8`; for UCS-2 and UTF-16 it would be `UInt16`; for UTF-32 it would be `UInt32`. The code unit type need not be limited to these three types, but it's hard to think of widely used string encodings that don't use one of these units. `codeunit(s)` is the same as `typeof(codeunit(s,1))` when `s` is a non-empty string.

See also `ncodeunits`.

source

`Base.codeunits` – Function.

```
codeunits(s::AbstractString)
```

Obtain a vector-like object containing the code units of a string. Returns a `CodeUnits` wrapper by default, but `codeunits` may optionally be defined for new string types if necessary.

Examples

```
julia> codeunits("Juλia")
6-element Base.CodeUnits{UInt8, String}:
 0x4a
 0x75
 0xce
 0xbb
 0x69
 0x61
```

source

`Base.ascii` – Function.

```
ascii(s::AbstractString)
```

Convert a string to `String` type and check that it contains only ASCII data, otherwise throwing an `ArgumentError` indicating the position of the first non-ASCII byte.

See also the `isascii` predicate to filter or replace non-ASCII characters.

Examples

```
julia> ascii("abcdeγfgh")
ERROR: ArgumentError: invalid ASCII at index 6 in "abcdeγfgh"
Stacktrace:
[...]

julia> ascii("abcdefgh")
"abcdefgh"
```

source

`Base.Regex` – Type.

```
Regex(pattern[, flags]) <: AbstractPattern
```

A type representing a regular expression. Regex objects can be used to match strings with `match`.

Regex objects can be created using the `@r_str` string macro. The `Regex(pattern[, flags])` constructor is usually used if the `pattern` string needs to be interpolated. See the documentation of the string macro for details on flags.

Note

To escape interpolated variables use `\Q` and `\E` (e.g. `Regex("\\Q$x\\E")`)

source

Base.@r_str – Macro.

```
@r_str -> Regex
```

Construct a regex, such as r"^[a-z]*$", without interpolation and unescaping (except for quotation mark " which still has to be escaped). The regex also accepts one or more flags, listed after the ending quote, to change its behaviour:

- i enables case-insensitive matching
- m treats the ^ and $ tokens as matching the start and end of individual lines, as opposed to the whole string.
- s allows the . modifier to match newlines.
- x enables "free-spacing mode": whitespace between regex tokens is ignored except when escaped with \, and # in the regex is treated as starting a comment (which is ignored to the line ending).
- a enables ASCII mode (disables UTF and UCP modes). By default \B, \b, \D, \d, \S, \s, \W, \w, etc. match based on Unicode character properties. With this option, these sequences only match ASCII characters. This includes \u also, which will emit the specified character value directly as a single byte, and not attempt to encode it into UTF-8. Importantly, this option allows matching against invalid UTF-8 strings, by treating both matcher and target as simple bytes (as if they were ISO/IEC 8859-1 / Latin-1 bytes) instead of as character encodings. In this case, this option is often combined with s. This option can be further refined by starting the pattern with (*UCP) or (*UTF).

See Regex if interpolation is needed.

Examples

```
julia> match(r"a+.*b+.*?d$"ism, "Goodbye,\nOh, angry,\nBad world\n")
RegexMatch("angry,\nBad world")
```

This regex has the first three flags enabled.

source

Base.SubstitutionString – Type.

```
SubstitutionString(substr) <: AbstractString
```

Stores the given string substr as a SubstitutionString, for use in regular expression substitutions. Most commonly constructed using the @s_str macro.

Examples

```
julia> SubstitutionString("Hello \\g<name>, it's \\1")
s"Hello \g<name>, it's \1"

julia> subst = s"Hello \g<name>, it's \1"
s"Hello \g<name>, it's \1"

julia> typeof(subst)
SubstitutionString{String}
```

source

Base.@s_str – Macro.

```
@s_str -> SubstitutionString
```

Construct a substitution string, used for regular expression substitutions. Within the string, sequences of the form \N refer to the Nth capture group in the regex, and \g<groupname> refers to a named capture group with name groupname.

Examples

```
julia> msg = "#Hello# from Julia";

julia> replace(msg, r"#(.+)# from (?<from>\w+)" => s"FROM: \g<from>; MESSAGE: \1")
"FROM: Julia; MESSAGE: Hello"
```

source

Base.@raw_str – Macro.

```
@raw_str -> String
```

Create a raw string without interpolation and unescaping. The exception is that quotation marks still must be escaped. Backslashes escape both quotation marks and other backslashes, but only when a sequence of backslashes precedes a quote character. Thus, 2n backslashes followed by a quote encodes n backslashes and the end of the literal while 2n+1 backslashes followed by a quote encodes n backslashes followed by a quote character.

Examples

```
julia> println(raw"\ $x")
\ $x

julia> println(raw"\"")
"

julia> println(raw"\\\"")
\"

julia> println(raw"\\x \\\"")
\\x \"
```

source

Base.@b_str – Macro.

```
@b_str
```

Create an immutable byte (UInt8) vector using string syntax.

Examples

```
julia> v = b"12\x01\x02"
4-element Base.CodeUnits{UInt8, String}:
 0x31
 0x32
 0x01
 0x02

julia> v[2]
0x32
```

source

Base.takestring! – Function.

```
takestring!(io::IOBuffer)::String
```

Return the content of `io` as a `String`, resetting the buffer to its initial state. This is preferred over calling `String(take!(io))` to create a string from an `IOBuffer`.

Examples

```
julia> io = IOBuffer();

julia> write(io, [0x61, 0x62, 0x63]);

julia> s = takestring!(io)
"abc"

julia> isempty(take!(io)) # io is now empty
true
```

Julia 1.13

This function requires at least Julia 1.13.

source

```
takestring!(x)::AbstractString
```

Create a string from the content of x, emptying x.

Examples

```
julia> v = [0x61, 0x62, 0x63];

julia> s = takestring!(v)
"abc"

julia> isempty(v)
true
```

Julia 1.13

This function requires at least Julia 1.13.

source

Base.Docs.@html_str – Macro.

```
@html_str -> Docs.HTML
```

Create an HTML object from a literal string.

Examples

```
julia> html"Julia"
HTML{String}("Julia")
```

source

Base.Docs.@text_str – Macro.

```
@text_str -> Docs.Text
```

Create a Text object from a literal string.

Examples

```
julia> text"Julia"
Julia
```

source

Base.isvalid – Method.

```
isvalid(value)::Bool
```

Return true if the given value is valid for its type, which currently can be either AbstractChar or String or SubString{String}.

Examples

```
julia> isvalid(Char(0xd800))
false

julia> isvalid(SubString(String(UInt8[0xfe,0x80,0x80,0x80,0x80,0x80]),1,2))
false

julia> isvalid(Char(0xd799))
true
```

source

Base.isvalid – Method.

```
isvalid(T, value)::Bool
```

Return true if the given value is valid for that type. Types currently can be either AbstractChar or String. Values for AbstractChar can be of type AbstractChar or UInt32. Values for String can be of that type, SubString{String}, Vector{UInt8}, or a contiguous subarray thereof.

Examples

```
julia> isvalid(Char, 0xd800)
false

julia> isvalid(String, SubString("thisisvalid",1,5))
true

julia> isvalid(Char, 0xd799)
true
```

Julia 1.6

Support for subarray values was added in Julia 1.6.

source

Base.isvalid – Method.

```
isvalid(s::AbstractString, i::Integer)::Bool
```

Predicate indicating whether the given index is the start of the encoding of a character in `s` or not. If `isvalid(s, i)` is true then `s[i]` will return the character whose encoding starts at that index, if it's false, then `s[i]` will raise an invalid index error or a bounds error depending on if `i` is in bounds. In order for `isvalid(s, i)` to be an O(1) function, the encoding of `s` must be self-synchronizing. This is a basic assumption of Julia's generic string support.

See also `getindex`, `iterate`, `thisind`, `nextind`, `prevind`, `length`.

Examples

```
julia> str = "αβγdef";

julia> isvalid(str, 1)
true

julia> str[1]
'α': Unicode U+03B1 (category Ll: Letter, lowercase)

julia> isvalid(str, 2)
false

julia> str[2]
ERROR: StringIndexError: invalid index [2], valid nearby indices [1]=>'α', [3]=>'β'
Stacktrace:
[...]
```

source

Base.match – Function.

```
match(r::Regex, s::AbstractString[, idx::Integer[, addopts]])
```

Search for the first match of the regular expression `r` in `s` and return a `RegexMatch` object containing the match, or nothing if the match failed. The optional `idx` argument specifies an index at which to start the search. The matching substring can be retrieved by accessing `m.match`, the captured sequences

can be retrieved by accessing `m.captures`. The resulting `RegexMatch` object can be used to construct other collections: e.g. `Tuple(m)`, `NamedTuple(m)`.

Julia 1.11

Constructing NamedTuples and Dicts requires Julia 1.11

Examples

```
julia> rx = r"a(.)a"
r"a(.)a"

julia> m = match(rx, "cabac")
RegexMatch("aba", 1="b")

julia> m.captures
1-element Vector{Union{Nothing, SubString{String}}}:
 "b"

julia> m.match
"aba"

julia> match(rx, "cabac", 3) === nothing
true
```

See also

`eachmatch`, `occursin`, `findfirst`

source

`Base.eachmatch` – Function.

```
eachmatch(r::Regex, s::AbstractString; overlap::Bool=false)
```

Search for all matches of the regular expression `r` in `s` and return an iterator over the matches. If `overlap` is `true`, the matching sequences are allowed to overlap indices in the original string, otherwise they must be from distinct character ranges.

Examples

```
julia> rx = r"a.a"
r"a.a"

julia> m = eachmatch(rx, "a1a2a3a")
Base.RegexMatchIterator{String}(r"a.a", "a1a2a3a", false)

julia> collect(m)
2-element Vector{RegexMatch}:
 RegexMatch("a1a")
 RegexMatch("a3a")

julia> collect(eachmatch(rx, "a1a2a3a", overlap = true))
3-element Vector{RegexMatch}:
```

```
RegexMatch("a1a")
RegexMatch("a2a")
RegexMatch("a3a")
```

See also

`match`, `findall`, `count`

`source`

`Base.RegexMatch` – Type.

```
RegexMatch <: AbstractMatch
```

A type representing a single match to a `Regex` found in a string. Typically created from the `match` function.

The `match` field stores the substring of the entire matched string. The `captures` field stores the substrings for each capture group, indexed by number. To index by capture group name, the entire match object should be indexed instead, as shown in the examples. The location of the start of the match is stored in the `offset` field. The `offsets` field stores the locations of the start of each capture group, with 0 denoting a group that was not captured.

This type can be used as an iterator over the capture groups of the Regex, yielding the substrings captured in each group. Because of this, the captures of a match can be destructured. If a group was not captured, `nothing` will be yielded instead of a substring.

Methods that accept a `RegexMatch` object are defined for `iterate`, `length`, `eltype`, `keys`, `haskey`, and `getindex`, where keys are the names or numbers of a capture group. See `keys` for more information.

`Tuple(m)`, `NamedTuple(m)`, and `Dict(m)` can be used to construct more flexible collection types from `RegexMatch` objects.

Julia 1.11

Constructing NamedTuples and Dicts from RegexMatches requires Julia 1.11

Examples

```
julia> m = match(r"(?<hour>\d+):(?<minute>\d+)(am|pm)?", "11:30 in the morning")
RegexMatch("11:30", hour="11", minute="30", 3=nothing)

julia> m.match
"11:30"

julia> m.captures
3-element Vector{Union{Nothing, SubString{String}}}:
 "11"
 "30"
 nothing


julia> m["minute"]
"30"
```

```
julia> hr, min, ampm = m; # destructure capture groups by iteration

julia> hr
"11"

julia> Dict(m)
Dict{Any, Union{Nothing, SubString{String}}} with 3 entries:
  "hour"   => "11"
  3        => nothing
  "minute" => "30"
```

source

Base.keys – Method.

```
keys(m::RegexMatch)::Vector
```

Return a vector of keys for all capture groups of the underlying regex. A key is included even if the capture group fails to match. That is, idx will be in the return value even if m[idx] == nothing.

Unnamed capture groups will have integer keys corresponding to their index. Named capture groups will have string keys.

Julia 1.7

This method was added in Julia 1.7

Examples

```
julia> keys(match(r"(?<hour>\d+):(?<minute>\d+)(am|pm)?", "11:30"))
3-element Vector{Any}:
  "hour"
  "minute"
 3
```

source

Base.isless – Method.

```
isless(a::AbstractString, b::AbstractString)::Bool
```

Test whether string a comes before string b in alphabetical order (technically, in lexicographical order by Unicode code points).

Examples

```
julia> isless("a", "b")
true

julia> isless("β", "α")
false

julia> isless("a", "a")
false
```

source

Base.:(==) – Method.

```
==(a::AbstractString, b::AbstractString)::Bool
```

Test whether two strings are equal character by character (technically, Unicode code point by code point). Should either string be a AnnotatedString the string properties must match too.

Examples

```
julia> "abc" == "abc"
true

julia> "abc" == "αβγ"
false
```

source

Base.cmp – Method.

```
cmp(a::AbstractString, b::AbstractString)::Int
```

Compare two strings. Return 0 if both strings have the same length and the character at each index is the same in both strings. Return -1 if a is a prefix of b, or if a comes before b in alphabetical order. Return 1 if b is a prefix of a, or if b comes before a in alphabetical order (technically, lexicographical order by Unicode code points).

Examples

```
julia> cmp("abc", "abc")
0

julia> cmp("ab", "abc")
-1

julia> cmp("abc", "ab")
1

julia> cmp("ab", "ac")
-1

julia> cmp("ac", "ab")
1

julia> cmp("α", "a")
1

julia> cmp("b", "β")
-1
```

source

Base.lpad – Function.

```
lpad(s, n::Integer, p::Union{AbstractChar,AbstractString}=' ')::String
```

Stringify `s` and pad the resulting string on the left with `p` to make it `n` characters (in `textwidth`) long. If `s` is already `n` characters long, an equal string is returned. Pad with spaces by default.

Examples

```
julia> lpad("March", 10)
"     March"
```

Julia 1.7

In Julia 1.7, this function was changed to use `textwidth` rather than a raw character (codepoint) count.

source

`Base.rpad` – Function.

```
rpad(s, n::Integer, p::Union{AbstractChar,AbstractString}=' ')::String
```

Stringify `s` and pad the resulting string on the right with `p` to make it `n` characters (in `textwidth`) long. If `s` is already `n` characters long, an equal string is returned. Pad with spaces by default.

Examples

```
julia> rpad("March", 20)
"March               "
```

Julia 1.7

In Julia 1.7, this function was changed to use `textwidth` rather than a raw character (codepoint) count.

source

`Base.ltruncate` – Function.

```
ltruncate(str::AbstractString, maxwidth::Integer,
↪  replacement::Union{AbstractString,AbstractChar} = '…')
```

Truncate `str` to at most `maxwidth` columns (as estimated by `textwidth`), replacing the first characters with `replacement` if necessary. The default replacement string is "…".

Examples

```
julia> s = ltruncate("□□ I love □", 10)
"…I love □"

julia> textwidth(s)
10

julia> ltruncate("foo", 3)
"foo"
```

Julia 1.12

This function was added in Julia 1.12.

See also `rtruncate` and `ctruncate`.

source

`Base.rtruncate` – Function.

```
rtruncate(str::AbstractString, maxwidth::Integer,
↪  replacement::Union{AbstractString,AbstractChar} = '…')
```

Truncate `str` to at most `maxwidth` columns (as estimated by `textwidth`), replacing the last characters with `replacement` if necessary. The default replacement string is "…".

Examples

```
julia> s = rtruncate("□□ I love □", 10)
"□□ I lo…"

julia> textwidth(s)
10

julia> rtruncate("foo", 3)
"foo"
```

Julia 1.12

This function was added in Julia 1.12.

See also `ltruncate` and `ctruncate`.

source

`Base.ctruncate` – Function.

```
ctruncate(str::AbstractString, maxwidth::Integer,
↪  replacement::Union{AbstractString,AbstractChar} = '…'; prefer_left::Bool = true)
```

Truncate `str` to at most `maxwidth` columns (as estimated by `textwidth`), replacing the middle characters with `replacement` if necessary. The default replacement string is "…". By default, the truncation prefers keeping chars on the left, but this can be changed by setting `prefer_left` to `false`.

Examples

```
julia> s = ctruncate("□□ I love □", 10)
"□□ …e □"

julia> textwidth(s)
10

julia> ctruncate("foo", 3)
"foo"
```

Julia 1.12

This function was added in Julia 1.12.

See also `ltruncate` and `rtruncate`.

source

Base.findfirst – Method.

```
findfirst(pattern::AbstractString, string::AbstractString)
findfirst(pattern::AbstractPattern, string::String)
```

Find the first occurrence of `pattern` in `string`. Equivalent to `findnext(pattern, string, firstindex(s))`.

Examples

```
julia> findfirst("z", "Hello to the world") # returns nothing, but not printed in the REPL

julia> findfirst("Julia", "JuliaLang")
1:5
```

source

Base.findnext – Method.

```
findnext(pattern::AbstractString, string::AbstractString, start::Integer)
findnext(pattern::AbstractPattern, string::String, start::Integer)
```

Find the next occurrence of `pattern` in `string` starting at position `start`. `pattern` can be either a string, or a regular expression, in which case `string` must be of type `String`.

The return value is a range of indices where the matching sequence is found, such that `s[findnext(x, s, i)] == x`:

`findnext("substring", string, i) == start:stop` such that `string[start:stop] == "substring"` and `i <= start`, or `nothing` if unmatched.

Examples

```
julia> findnext("z", "Hello to the world", 1) === nothing
true

julia> findnext("o", "Hello to the world", 6)
8:8

julia> findnext("Lang", "JuliaLang", 2)
6:9
```

source

Base.findnext – Method.

```
findnext(ch::AbstractChar, string::AbstractString, start::Integer)
```

Find the next occurrence of character `ch` in `string` starting at position `start`.

Julia 1.3

This method requires at least Julia 1.3.

Examples

```
julia> findnext('z', "Hello to the world", 1) === nothing
true

julia> findnext('o', "Hello to the world", 6)
8
```

source

`Base.findlast` – Method.

```
findlast(pattern::AbstractString, string::AbstractString)
```

Find the last occurrence of `pattern` in `string`. Equivalent to `findprev(pattern, string, lastindex(string))`.

Examples

```
julia> findlast("o", "Hello to the world")
15:15

julia> findfirst("Julia", "JuliaLang")
1:5
```

source

`Base.findlast` – Method.

```
findlast(ch::AbstractChar, string::AbstractString)
```

Find the last occurrence of character `ch` in `string`.

Julia 1.3

This method requires at least Julia 1.3.

Examples

```
julia> findlast('p', "happy")
4

julia> findlast('z', "happy") === nothing
true
```

source

`Base.findprev` – Method.

```
findprev(pattern::AbstractString, string::AbstractString, start::Integer)
```

Find the previous occurrence of `pattern` in `string` starting at position `start`.

The return value is a range of indices where the matching sequence is found, such that `s[findprev(x, s, i)] == x`:

`findprev("substring", string, i)` == `start:stop` such that `string[start:stop] == "substring"` and `stop <= i`, or `nothing` if unmatched.

Examples

```
julia> findprev("z", "Hello to the world", 18) === nothing
true

julia> findprev("o", "Hello to the world", 18)
15:15

julia> findprev("Julia", "JuliaLang", 6)
1:5
```

source

`Base.occursin` – Function.

```
occursin(haystack)
```

Create a function that checks whether its argument occurs in `haystack`, i.e. a function equivalent to `needle -> occursin(needle, haystack)`.

The returned function is of type `Base.Fix2{typeof(occursin)}`.

Julia 1.6

This method requires Julia 1.6 or later.

Examples

```
julia> search_f = occursin("JuliaLang is a programming language");

julia> search_f("JuliaLang")
true

julia> search_f("Python")
false
```

source

```
occursin(needle::Union{AbstractString,AbstractPattern,AbstractChar}, haystack::AbstractString)
```

Determine whether the first argument is a substring of the second. If `needle` is a regular expression, checks whether `haystack` contains a match.

Examples

```
julia> occursin("Julia", "JuliaLang is pretty cool!")
true

julia> occursin('a', "JuliaLang is pretty cool!")
true

julia> occursin(r"a.a", "aba")
true

julia> occursin(r"a.a", "abba")
false
```

See also `contains`.

source

Base.reverse – Method.

```
reverse(s::AbstractString)::AbstractString
```

Reverses a string. Technically, this function reverses the codepoints in a string and its main utility is for reversed-order string processing, especially for reversed regular-expression searches. See also `reverseind` to convert indices in `s` to indices in `reverse(s)` and vice-versa, and `graphemes` from module `Unicode` to operate on user-visible "characters" (graphemes) rather than codepoints. See also `Iterators.reverse` for reverse-order iteration without making a copy. Custom string types must implement the `reverse` function themselves and should typically return a string with the same type and encoding. If they return a string with a different encoding, they must also override `reverseind` for that string type to satisfy `s[reverseind(s,i)] == reverse(s)[i]`.

Examples

```
julia> reverse("JuliaLang")
"gnaLailuJ"
```

Note

The examples below may be rendered differently on different systems. The comments indicate how they're supposed to be rendered

Combining characters can lead to surprising results:

```
julia> reverse("ax̂e") # hat is above x in the input, above e in the output
"êxa"

julia> using Unicode

julia> join(reverse(collect(graphemes("ax̂e")))) # reverses graphemes; hat is above x in both
↪  in- and output
"ex̂a"
```

source

Base.replace – Method.

```
replace([io::IO], s::AbstractString, pat=>r, [pat2=>r2, ...]; [count::Integer])
```

Search for the given pattern `pat` in `s`, and replace each occurrence with `r`. If `count` is provided, replace at most `count` occurrences. `pat` may be a single character, a vector or a set of characters, a string, or a regular expression. If `r` is a function, each occurrence is replaced with `r(s)` where `s` is the matched substring (when `pat` is a `AbstractPattern` or `AbstractString`) or character (when `pat` is an `AbstractChar` or a collection of `AbstractChar`). If `pat` is a regular expression and `r` is a `SubstitutionString`, then capture group references in `r` are replaced with the corresponding matched text. To remove instances of `pat` from `string`, set `r` to the empty `String` (`""`).

The return value is a new string after the replacements. If the `io::IO` argument is supplied, the transformed string is instead written to `io` (returning `io`). (For example, this can be used in conjunction with an `IOBuffer` to re-use a pre-allocated buffer array in-place.)

Multiple patterns can be specified: The input string will be scanned only once from start (left) to end (right), and the first matching replacement will be applied to each substring. Replacements are applied in the order of the arguments provided if they match substrings starting at the same input string position. Thus, only one pattern will be applied to any character, and the patterns will only be applied to the input text, not the replacements.

Julia 1.7

Support for multiple patterns requires version 1.7.

Julia 1.10

The `io::IO` argument requires version 1.10.

Examples

```
julia> replace("Python is a programming language.", "Python" => "Julia")
"Julia is a programming language."

julia> replace("The quick foxes run quickly.", "quick" => "slow", count=1)
"The slow foxes run quickly."

julia> replace("The quick foxes run quickly.", "quick" => "", count=1)
"The  foxes run quickly."

julia> replace("The quick foxes run quickly.", r"fox(es)?" => s"bus\1")
"The quick buses run quickly."

julia> replace("abcabc", "a" => "b", "b" => "c", r".+" => "a")
"bca"
```

source

`Base.eachsplit` – Function.

```
eachsplit(str::AbstractString, dlm; limit::Integer=0, keepempty::Bool=true)
eachsplit(str::AbstractString; limit::Integer=0, keepempty::Bool=false)
```

Split `str` on occurrences of the delimiter(s) `dlm` and return an iterator over the substrings. `dlm` can be any of the formats allowed by `findnext`'s first argument (i.e. as a string, regular expression or a function), or as a single character or collection of characters.

If `dlm` is omitted, it defaults to `isspace`.

The optional keyword arguments are:

- `limit`: the maximum size of the result. `limit=0` implies no maximum (default)
- `keepempty`: whether empty fields should be kept in the result. Default is `false` without a `dlm` argument, `true` with a `dlm` argument.

See also `split`.

Julia 1.8

The `eachsplit` function requires at least Julia 1.8.

Examples

```
julia> a = "Ma.rch"
"Ma.rch"

julia> b = eachsplit(a, ".")
Base.SplitIterator{String, String}("Ma.rch", ".", 0, true)

julia> collect(b)
2-element Vector{SubString{String}}:
 "Ma"
 "rch"
```

source

`Base.eachrsplit` – Function.

```
eachrsplit(str::AbstractString, dlm; limit::Integer=0, keepempty::Bool=true)
eachrsplit(str::AbstractString; limit::Integer=0, keepempty::Bool=false)
```

Return an iterator over SubStrings of `str`, produced when splitting on the delimiter(s) `dlm`, and yielded in reverse order (from right to left). `dlm` can be any of the formats allowed by `findprev`'s first argument (i.e. a string, a single character or a function), or a collection of characters.

If `dlm` is omitted, it defaults to `isspace`, and keepempty default to `false`.

The optional keyword arguments are:

- If `limit > 0`, the iterator will split at most `limit - 1` times before returning the rest of the string unsplit. `limit < 1` implies no cap to splits (default).
- `keepempty`: whether empty fields should be returned when iterating Default is `false` without a `dlm` argument, `true` with a `dlm` argument.

Note that unlike `split`, `rsplit` and `eachsplit`, this function iterates the substrings right to left as they occur in the input.

See also `eachsplit`, `rsplit`.

Julia 1.11

This function requires Julia 1.11 or later.

Examples

```
julia> a = "Ma.r.ch";

julia> collect(eachrsplit(a, ".")) == ["ch", "r", "Ma"]
true

julia> collect(eachrsplit(a, "."; limit=2)) == ["ch", "Ma.r"]
true
```

source

Base.split – Function.

```
split(str::AbstractString, dlm; limit::Integer=0, keepempty::Bool=true)
split(str::AbstractString; limit::Integer=0, keepempty::Bool=false)
```

Split `str` into an array of substrings on occurrences of the delimiter(s) `dlm`. `dlm` can be any of the formats allowed by `findnext`'s first argument (i.e. as a string, regular expression or a function), or as a single character or collection of characters.

If `dlm` is omitted, it defaults to `isspace`.

The optional keyword arguments are:

- `limit`: the maximum size of the result. `limit=0` implies no maximum (default)
- `keepempty`: whether empty fields should be kept in the result. Default is `false` without a `dlm` argument, `true` with a `dlm` argument.

See also `rsplit`, `eachsplit`.

Examples

```
julia> a = "Ma.rch"
"Ma.rch"

julia> split(a, ".")
2-element Vector{SubString{String}}:
 "Ma"
 "rch"
```

source

Base.rsplit – Function.

```
rsplit(s::AbstractString; limit::Integer=0, keepempty::Bool=false)
rsplit(s::AbstractString, chars; limit::Integer=0, keepempty::Bool=true)
```

Similar to `split`, but starting from the end of the string.

Examples

```
julia> a = "M.a.r.c.h"
"M.a.r.c.h"

julia> rsplit(a, ".")
5-element Vector{SubString{String}}:
 "M"
 "a"
 "r"
 "c"
 "h"

julia> rsplit(a, "."; limit=1)
1-element Vector{SubString{String}}:
 "M.a.r.c.h"

julia> rsplit(a, "."; limit=2)
2-element Vector{SubString{String}}:
 "M.a.r.c"
 "h"
```

source

Base.strip – Function.

```
strip([pred=isspace,] str::AbstractString)::SubString
strip(str::AbstractString, chars)::SubString
```

Remove leading and trailing characters from `str`, either those specified by `chars` or those for which the function `pred` returns `true`.

The default behaviour is to remove leading and trailing whitespace and delimiters: see `isspace` for precise details.

The optional `chars` argument specifies which characters to remove: it can be a single character, vector or set of characters.

See also `lstrip` and `rstrip`.

Julia 1.2

The method which accepts a predicate function requires Julia 1.2 or later.

Examples

```
julia> strip("{3, 5}\n", ['{', '}', '\n'])
"3, 5"
```

source

Base.lstrip – Function.

```
lstrip([pred=isspace,] str::AbstractString)::SubString
lstrip(str::AbstractString, chars)::SubString
```

Remove leading characters from `str`, either those specified by `chars` or those for which the function `pred` returns `true`.

The default behaviour is to remove leading whitespace and delimiters: see `isspace` for precise details.

The optional `chars` argument specifies which characters to remove: it can be a single character, or a vector or set of characters.

See also `strip` and `rstrip`.

Examples

```
julia> a = lpad("March", 20)
"               March"

julia> lstrip(a)
"March"
```

source

Base.rstrip – Function.

```
rstrip([pred=isspace,] str::AbstractString)::SubString
rstrip(str::AbstractString, chars)::SubString
```

Remove trailing characters from `str`, either those specified by `chars` or those for which the function `pred` returns `true`.

The default behaviour is to remove trailing whitespace and delimiters: see `isspace` for precise details.

The optional `chars` argument specifies which characters to remove: it can be a single character, or a vector or set of characters.

See also `strip` and `lstrip`.

Examples

```
julia> a = rpad("March", 20)
"March               "

julia> rstrip(a)
"March"
```

source

Base.startswith – Function.

```
startswith(s::AbstractString, prefix::Regex)
```

Return `true` if `s` starts with the regex pattern, `prefix`.

Note

`startswith` does not compile the anchoring into the regular expression, but instead passes the anchoring as `match_option` to PCRE. If compile time is amortized, `occursin(r"^...", s)` is faster than `startswith(s, r"...")`.

See also `occursin`, `endswith`, `match`

Julia 1.2

This method requires at least Julia 1.2.

Examples

```
julia> startswith("JuliaLang", r"Julia|Romeo")
true
```

source

```
startswith(prefix)
```

Create a function that checks whether its argument starts with `prefix`, i.e. a function equivalent to `y -> startswith(y, prefix)`.

The returned function is of type `Base.Fix2{typeof(startswith)}`, which can be used to implement specialized methods.

Julia 1.5

The single argument `startswith(prefix)` requires at least Julia 1.5.

Examples

```
julia> startswith("Julia")("JuliaLang")
true

julia> startswith("Julia")("Ends with Julia")
false
```

source

```
startswith(io::IO, prefix::Union{AbstractString,Base.Chars})
```

Check if an `IO` object starts with a prefix, which can be either a string, a character, or a tuple/vector/set of characters. See also `peek`.

source

```
startswith(s::AbstractString, prefix::Union{AbstractString,Base.Chars})
```

Return `true` if `s` starts with `prefix`, which can be a string, a character, or a tuple/vector/set of characters. If `prefix` is a tuple/vector/set of characters, test whether the first character of `s` belongs to that set.

See also `endswith`, `contains`.

Examples

```
julia> startswith("JuliaLang", "Julia")
true
```

source

Base.endswith – Function.

```
endswith(s::AbstractString, suffix::Regex)
```

Return `true` if `s` ends with the regex pattern, `suffix`.

Note

`endswith` does not compile the anchoring into the regular expression, but instead passes the anchoring as `match_option` to PCRE. If compile time is amortized, `occursin(r"...$", s)` is faster than `endswith(s, r"...")`.

See also `occursin`, `startswith`, `match`

Julia 1.2

This method requires at least Julia 1.2.

Examples

```
julia> endswith("JuliaLang", r"Lang|Roberts")
true
```

source

```
endswith(suffix)
```

Create a function that checks whether its argument ends with `suffix`, i.e. a function equivalent to `y -> endswith(y, suffix)`.

The returned function is of type `Base.Fix2{typeof(endswith)}`, which can be used to implement specialized methods.

Julia 1.5

The single argument `endswith(suffix)` requires at least Julia 1.5.

Examples

```
julia> endswith("Julia")("Ends with Julia")
true

julia> endswith("Julia")("JuliaLang")
false
```

source

```
endswith(s::AbstractString, suffix::Union{AbstractString,Base.Chars})
```

Return `true` if `s` ends with `suffix`, which can be a string, a character, or a tuple/vector/set of characters. If `suffix` is a tuple/vector/set of characters, test whether the last character of `s` belongs to that set.

See also `startswith`, `contains`.

Examples

```
julia> endswith("Sunday", "day")
true
```

source

`Base.contains` – Function.

```
contains(needle)
```

Create a function that checks whether its argument contains `needle`, i.e. a function equivalent to `haystack -> contains(haystack, needle)`.

The returned function is of type `Base.Fix2{typeof(contains)}`, which can be used to implement specialized methods.

source

```
contains(haystack::AbstractString, needle)
```

Return `true` if `haystack` contains `needle`. This is the same as `occursin(needle, haystack)`, but is provided for consistency with `startswith(haystack, needle)` and `endswith(haystack, needle)`.

See also `occursin`, `in`, `issubset`.

Examples

```
julia> contains("JuliaLang is pretty cool!", "Julia")
true

julia> contains("JuliaLang is pretty cool!", 'a')
true

julia> contains("aba", r"a.a")
true

julia> contains("abba", r"a.a")
false
```

Julia 1.5

The `contains` function requires at least Julia 1.5.

source

`Base.first` – Method.

```
first(s::AbstractString, n::Integer)
```

Get a string consisting of the first `n` characters of `s`.

Examples

```
julia> first("∀ϵ≠0: ϵ²>0", 0)
""

julia> first("∀ϵ≠0: ϵ²>0", 1)
"∀"

julia> first("∀ϵ≠0: ϵ²>0", 3)
"∀ϵ≠"
```

source

`Base.last` – Method.

```
last(s::AbstractString, n::Integer)
```

Get a string consisting of the last `n` characters of `s`.

Examples

```
julia> last("∀ϵ≠0: ϵ²>0", 0)
""

julia> last("∀ϵ≠0: ϵ²>0", 1)
"0"

julia> last("∀ϵ≠0: ϵ²>0", 3)
"²>0"
```

source

`Base.Unicode.uppercase` – Function.

```
uppercase(s::AbstractString)
```

Return `s` with all characters converted to uppercase.

See also `lowercase`, `titlecase`, `uppercasefirst`.

Examples

```
julia> uppercase("Julia")
"JULIA"
```

source

```
uppercase(c::AbstractChar)
```

Convert c to uppercase.

See also lowercase, titlecase.

Examples

```
julia> uppercase('a')
'A': ASCII/Unicode U+0041 (category Lu: Letter, uppercase)

julia> uppercase('ê')
'Ê': Unicode U+00CA (category Lu: Letter, uppercase)
```

source

Base.Unicode.lowercase – Function.

```
lowercase(s::AbstractString)
```

Return s with all characters converted to lowercase.

See also uppercase, titlecase, lowercasefirst.

Examples

```
julia> lowercase("STRINGS AND THINGS")
"strings and things"
```

source

```
lowercase(c::AbstractChar)
```

Convert c to lowercase.

See also uppercase, titlecase.

Examples

```
julia> lowercase('A')
'a': ASCII/Unicode U+0061 (category Ll: Letter, lowercase)

julia> lowercase('Ö')
'ö': Unicode U+00F6 (category Ll: Letter, lowercase)
```

source

Base.Unicode.titlecase – Function.

```
titlecase(s::AbstractString; [wordsep::Function], strict::Bool=true)::String
```

Capitalize the first character of each word in `s`; if `strict` is true, every other character is converted to lowercase, otherwise they are left unchanged. By default, all non-letters beginning a new grapheme are considered as word separators; a predicate can be passed as the `wordsep` keyword to determine which characters should be considered as word separators. See also `uppercasefirst` to capitalize only the first character in `s`.

See also `uppercase`, `lowercase`, `uppercasefirst`.

Examples

```
julia> titlecase("the JULIA programming language")
"The Julia Programming Language"

julia> titlecase("ISS - international space station", strict=false)
"ISS - International Space Station"

julia> titlecase("a-a b-b", wordsep = c->c==' ')
"A-a B-b"
```

source

```
titlecase(c::AbstractChar)
```

Convert `c` to titlecase. This may differ from uppercase for digraphs, compare the example below.

See also `uppercase`, `lowercase`.

Examples

```
julia> titlecase('a')
'A': ASCII/Unicode U+0041 (category Lu: Letter, uppercase)

julia> titlecase('ǆ')
'ǅ': Unicode U+01C5 (category Lt: Letter, titlecase)

julia> uppercase('ǆ')
'Ǆ': Unicode U+01C4 (category Lu: Letter, uppercase)
```

source

`Base.Unicode.uppercasefirst` – Function.

```
uppercasefirst(s::AbstractString)::String
```

Return `s` with the first character converted to uppercase (technically "title case" for Unicode). See also `titlecase` to capitalize the first character of every word in `s`.

See also `lowercasefirst`, `uppercase`, `lowercase`, `titlecase`.

Examples

```
julia> uppercasefirst("python")
"Python"
```

source

Base.Unicode.lowercasefirst – Function.

```
lowercasefirst(s::AbstractString)
```

Return `s` with the first character converted to lowercase.

See also `uppercasefirst`, `uppercase`, `lowercase`, `titlecase`.

Examples

```
julia> lowercasefirst("Julia")
"julia"
```

source

Base.join – Function.

```
join([io::IO,] iterator [, delim [, last]])
```

Join any `iterator` into a single string, inserting the given delimiter (if any) between adjacent items. If `last` is given, it will be used instead of `delim` between the last two items. Each item of `iterator` is converted to a string via `print(io::IOBuffer, x)`. If `io` is given, the result is written to `io` rather than returned as a `String`.

Examples

```
julia> join(["apples", "bananas", "pineapples"], ", ", " and ")
"apples, bananas and pineapples"

julia> join([1,2,3,4,5])
"12345"
```

source

Base.chop – Function.

```
chop(s::AbstractString; head::Integer = 0, tail::Integer = 1)
```

Remove the first `head` and the last `tail` characters from `s`. The call `chop(s)` removes the last character from `s`. If it is requested to remove more characters than `length(s)` then an empty string is returned.

See also `chomp`, `startswith`, `first`.

Examples

```
julia> a = "March"
"March"

julia> chop(a)
"Marc"

julia> chop(a, head = 1, tail = 2)
"ar"

julia> chop(a, head = 5, tail = 5)
""
```

source

Base.chopprefix – Function.

```
chopprefix(s::AbstractString, prefix::Union{AbstractString,Regex,AbstractChar})::SubString
```

Remove the prefix `prefix` from `s`. If `s` does not start with `prefix`, a string equal to `s` is returned.

See also chopsuffix.

Julia 1.8

This function is available as of Julia 1.8.

Julia 1.13

The method which accepts an `AbstractChar` prefix is available as of Julia 1.13.

Examples

```
julia> chopprefix("Hamburger", "Ham")
"burger"

julia> chopprefix("Hamburger", "hotdog")
"Hamburger"
```

source

Base.chopsuffix – Function.

```
chopsuffix(s::AbstractString, suffix::Union{AbstractString,Regex,AbstractChar})::SubString
```

Remove the suffix `suffix` from `s`. If `s` does not end with `suffix`, a string equal to `s` is returned.

See also choppref ix.

Julia 1.8

This function is available as of Julia 1.8.

Julia 1.13

The method which accepts an `AbstractChar` suffix is available as of Julia 1.13.

Examples

```
julia> chopsuffix("Hamburger", "er")
"Hamburg"

julia> chopsuffix("Hamburger", "hotdog")
"Hamburger"
```

source

Base.chomp – Function.

```
chomp(s::AbstractString)::SubString
```

Remove a single trailing newline (i.e. "\r\n" or "\n") from a string.

See also chop.

Examples

```
julia> chomp("Hello\n")
"Hello"

julia> chomp("World\r\n")
"World"

julia> chomp("Julia\r\n\n")
"Julia\r\n"
```

source

Base.thisind – Function.

```
thisind(s::AbstractString, i::Integer)::Int
```

If i is in bounds in s return the index of the start of the character whose encoding code unit i is part of. In other words, if i is the start of a character, return i; if i is not the start of a character, rewind until the start of a character and return that index. If i is equal to 0 or ncodeunits(s)+1 return i. In all other cases throw BoundsError.

Examples

```
julia> thisind("α", 0)
0

julia> thisind("α", 1)
1

julia> thisind("α", 2)
1

julia> thisind("α", 3)
3

julia> thisind("α", 4)
ERROR: BoundsError: attempt to access 2-codeunit String at index [4]
[...]

julia> thisind("α", -1)
ERROR: BoundsError: attempt to access 2-codeunit String at index [-1]
[...]
```

source

Base.nextind – Method.

```
nextind(str::AbstractString, i::Integer, n::Integer=1)::Int
```

- Case `n == 1`

  If `i` is in bounds in `str` return the index of the start of the character whose encoding starts after index `i`. In other words, if `i` is the start of a character, return the start of the next character; if `i` is not the start of a character, move forward until the start of a character and return that index. If `i` is equal to `0` return `1`. If `i` is in bounds but greater or equal to `lastindex(str)` return `ncodeunits(str)+1`. Otherwise throw `BoundsError`.

- Case `n > 1`

  Behaves like applying `n` times `nextind` for `n==1`. The only difference is that if `n` is so large that applying `nextind` would reach `ncodeunits(str)+1` then each remaining iteration increases the returned value by `1`. This means that in this case `nextind` can return a value greater than `ncodeunits(str)+1`.

- Case `n == 0`

  Return `i` only if `i` is a valid index in `str` or is equal to `0`. Otherwise `StringIndexError` or `BoundsError` is thrown.

Examples

```
julia> nextind("α", 0)
1

julia> nextind("α", 1)
3

julia> nextind("α", 3)
ERROR: BoundsError: attempt to access 2-codeunit String at index [3]
[...]

julia> nextind("α", 0, 2)
3

julia> nextind("α", 1, 2)
4
```

source

`Base.prevind` – Method.

```
prevind(str::AbstractString, i::Integer, n::Integer=1)::Int
```

- Case `n == 1`

  If `i` is in bounds in `str` return the index of the start of the character whose encoding starts before index `i`. In other words, if `i` is the start of a character, return the start of the previous character; if `i` is not the start of a character, rewind until the start of a character and return that index. If `i` is equal to `1` return `0`. If `i` is equal to `ncodeunits(str)+1` return `lastindex(str)`. Otherwise throw `BoundsError`.

- Case `n > 1`

  Behaves like applying `n` times `prevind` for `n==1`. The only difference is that if `n` is so large that applying `prevind` would reach `0` then each remaining iteration decreases the returned value by `1`. This means that in this case `prevind` can return a negative value.

- Case `n == 0`

  Return `i` only if `i` is a valid index in `str` or is equal to `ncodeunits(str)+1`. Otherwise `StringIndexError` or `BoundsError` is thrown.

Examples

```
julia> prevind("α", 3)
1

julia> prevind("α", 1)
0

julia> prevind("α", 0)
ERROR: BoundsError: attempt to access 2-codeunit String at index [0]
[...]

julia> prevind("α", 2, 2)
0

julia> prevind("α", 2, 3)
-1
```

source

`Base.Unicode.textwidth` – Function.

```
textwidth(s::AbstractString)
```

Give the number of columns needed to print a string.

Examples

```
julia> textwidth("March")
5
```

source

```
textwidth(c)
```

Give the number of columns needed to print a character.

Examples

```
julia> textwidth('α')
1

julia> textwidth('□')
2
```

source

`Base.isascii` – Function.

```
isascii(cu::AbstractVector{CU}) where {CU <: Integer}::Bool
```

Test whether all values in the vector belong to the ASCII character set (0x00 to 0x7f). This function is intended to be used by other string implementations that need a fast ASCII check.

source

```
isascii(c::Union{AbstractChar,AbstractString})::Bool
```

Test whether a character belongs to the ASCII character set, or whether this is true for all elements of a string.

Examples

```
julia> isascii('a')
true

julia> isascii('α')
false

julia> isascii("abc")
true

julia> isascii("αβγ")
false
```

For example, `isascii` can be used as a predicate function for `filter` or `replace` to remove or replace non-ASCII characters, respectively:

```
julia> filter(isascii, "abcdeγfgh") # discard non-ASCII chars
"abcdefgh"

julia> replace("abcdeγfgh", !isascii=>' ') # replace non-ASCII chars with spaces
"abcde fgh"
```

source

`Base.Unicode.iscntrl` – Function.

```
iscntrl(c::AbstractChar)::Bool
```

Tests whether a character is a control character. Control characters are the non-printing characters of the Latin-1 subset of Unicode.

Examples

```
julia> iscntrl('\x01')
true

julia> iscntrl('a')
false
```

source

`Base.Unicode.isdigit` – Function.

```
isdigit(c::AbstractChar)::Bool
```

Tests whether a character is an ASCII decimal digit (0-9).

See also: `isletter`.

Examples

```
julia> isdigit('❤')
false

julia> isdigit('9')
true

julia> isdigit('α')
false
```

source

`Base.Unicode.isletter` – Function.

```
isletter(c::AbstractChar)::Bool
```

Test whether a character is a letter. A character is classified as a letter if it belongs to the Unicode general category Letter, i.e. a character whose category code begins with 'L'.

See also: `isdigit`.

Examples

```
julia> isletter('❤')
false

julia> isletter('α')
true

julia> isletter('9')
false
```

source

`Base.Unicode.islowercase` – Function.

```
islowercase(c::AbstractChar)::Bool
```

Tests whether a character is a lowercase letter (according to the Unicode standard's `Lowercase` derived property).

See also `isuppercase`.

Examples

```
julia> islowercase('α')
true
```

```
julia> islowercase('Γ')
false

julia> islowercase('❤')
false
```

source

Base.Unicode.isnumeric – Function.

```
isnumeric(c::AbstractChar)::Bool
```

Tests whether a character is numeric. A character is classified as numeric if it belongs to the Unicode general category Number, i.e. a character whose category code begins with 'N'.

Note that this broad category includes characters such as ¾ and ৫. Use `isdigit` to check whether a character is a decimal digit between 0 and 9.

Examples

```
julia> isnumeric('৫')
true

julia> isnumeric('9')
true

julia> isnumeric('α')
false

julia> isnumeric('❤')
false
```

source

Base.Unicode.isprint – Function.

```
isprint(c::AbstractChar)::Bool
```

Tests whether a character is printable, including spaces, but not a control character.

Examples

```
julia> isprint('\x01')
false

julia> isprint('A')
true
```

source

Base.Unicode.ispunct – Function.

```
ispunct(c::AbstractChar)::Bool
```

Tests whether a character belongs to the Unicode general category Punctuation, i.e. a character whose category code begins with 'P'.

Note

This behavior is different from the `ispunct` function in C.

Examples

```
julia> ispunct('α')
false

julia> ispunct('=')
false

julia> ispunct('/')
true

julia> ispunct(';')
true
```

source

`Base.Unicode.isspace` – Function.

```
isspace(c::AbstractChar)::Bool
```

Tests whether a character is any whitespace character. Includes ASCII characters '\t', '\n', '\v', '\f', '\r', and ' ', Latin-1 character U+0085, and characters in Unicode category Zs.

Examples

```
julia> isspace('\n')
true

julia> isspace('\r')
true

julia> isspace(' ')
true

julia> isspace('\x20')
true
```

source

`Base.Unicode.isuppercase` – Function.

```
isuppercase(c::AbstractChar)::Bool
```

Tests whether a character is an uppercase letter (according to the Unicode standard's `Uppercase` derived property).

See also `islowercase`.

Examples

```
julia> isuppercase('γ')
false

julia> isuppercase('Γ')
true

julia> isuppercase('❤')
false
```

source

Base.Unicode.isxdigit – Function.

```
isxdigit(c::AbstractChar)::Bool
```

Test whether a character is a valid hexadecimal digit. Note that this does not include x (as in the standard 0x prefix).

Examples

```
julia> isxdigit('a')
true

julia> isxdigit('x')
false
```

source

Base.escape_string – Function.

```
escape_string(str::AbstractString[, esc]; keep=(), ascii=false, fullhex=false)::AbstractString
escape_string(io, str::AbstractString[, esc]; keep=())::Nothing
```

General escaping of traditional C and Unicode escape sequences. The first form returns the escaped string, the second prints the result to io.

Backslashes (\) are escaped with a double-backslash ("\\"). Non-printable characters are escaped either with their standard C escape codes, "\0" for NUL (if unambiguous), unicode code point ("\u" prefix) or hex ("\x" prefix).

The optional esc argument specifies any additional characters that should also be escaped by a prepending backslash (" is also escaped by default in the first form).

The argument keep specifies a collection of characters which are to be kept as they are. Notice that esc has precedence here.

The argument ascii can be set to true to escape all non-ASCII characters, whereas the default ascii=false outputs printable Unicode characters as-is. (keep takes precedence over ascii.)

The argument fullhex can be set to true to require all \u escapes to be printed with 4 hex digits, and \U escapes to be printed with 8 hex digits, whereas by default (fullhex=false) they are printed with fewer digits if possible (omitting leading zeros).

See also unescape_string for the reverse operation.

Julia 1.7

The keep argument is available as of Julia 1.7.

Julia 1.12

The ascii and fullhex arguments require Julia 1.12.

Examples

```
julia> escape_string("aaa\nbbb")
"aaa\\nbbb"

julia> escape_string("aaa\nbbb"; keep = '\n')
"aaa\nbbb"

julia> escape_string("\xfe\xff") # invalid utf-8
"\\xfe\\xff"

julia> escape_string(string('\u2135','\0')) # unambiguous
"ℵ\\0"

julia> escape_string(string('\u2135','\0','0')) # \0 would be ambiguous
"ℵ\\x000"
```

source

Base.escape_raw_string – Function.

```
escape_raw_string(s::AbstractString, delim='"')::AbstractString
escape_raw_string(io, s::AbstractString, delim='"')
```

Escape a string in the manner used for parsing raw string literals. For each double-quote (") character in input string s (or delim if specified), this function counts the number n of preceding backslash (\) characters, and then increases there the number of backslashes from n to $2n+1$ (even for $n = 0$). It also doubles a sequence of backslashes at the end of the string.

This escaping convention is used in raw strings and other non-standard string literals. (It also happens to be the escaping convention expected by the Microsoft C/C++ compiler runtime when it parses a command-line string into the argv[] array.)

See also Base.escape_string().

source

Base.unescape_string – Function.

```
unescape_string(str::AbstractString, keep = ())::AbstractString
unescape_string(io, s::AbstractString, keep = ())::Nothing
```

General unescaping of traditional C and Unicode escape sequences. The first form returns the escaped string, the second prints the result to io. The argument keep specifies a collection of characters which (along with backlashes) are to be kept as they are.

The following escape sequences are recognised:

- Escaped backslash (\\)
- Escaped double-quote (\")
- Standard C escape sequences (\a, \b, \t, \n, \v, \f, \r, \e)
- Unicode BMP code points (\u with 1-4 trailing hex digits)
- All Unicode code points (\U with 1-8 trailing hex digits; max value = 0010ffff)
- Hex bytes (\x with 1-2 trailing hex digits)
- Octal bytes (\ with 1-3 trailing octal digits)

See also `escape_string`.

Examples

```
julia> unescape_string("aaa\\nbbb") # C escape sequence
"aaa\nbbb"

julia> unescape_string("\\u03c0") # unicode
"π"

julia> unescape_string("\\101") # octal
"A"

julia> unescape_string("aaa \\g \\n", ['g']) # using `keep` argument
"aaa \\g \n"
```

source

48.1 AnnotatedStrings

Note

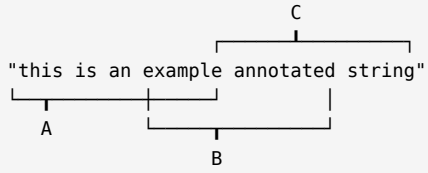
The API for AnnotatedStrings is considered experimental and is subject to change between Julia versions.

`Base.AnnotatedString` – Type.

```
AnnotatedString{S <: AbstractString} <: AbstractString
```

A string with metadata, in the form of annotated regions.

More specifically, this is a simple wrapper around any other `AbstractString` that allows for regions of the wrapped string to be annotated with labeled values.



The above diagram represents a `AnnotatedString` where three ranges have been annotated (labeled A, B, and C). Each annotation holds a label (`Symbol`) and a value (`Any`). These three pieces of information are held as a `@NamedTuple{region::UnitRange{Int64}, label::Symbol, value}`.

Labels do not need to be unique, the same region can hold multiple annotations with the same label.

Code written for `AnnotatedStrings` in general should conserve the following properties:

- Which characters an annotation is applied to
- The order in which annotations are applied to each character

Additional semantics may be introduced by specific uses of `AnnotatedStrings`.

A corollary of these rules is that adjacent, consecutively placed, annotations with identical labels and values are equivalent to a single annotation spanning the combined range.

See also `AnnotatedChar`, `annotatedstring`, `annotations`, and `annotate!`.

Constructors

```
AnnotatedString(s::S<:AbstractString) -> AnnotatedString{S}
AnnotatedString(s::S<:AbstractString,
↪  annotations::Vector{@NamedTuple{region::UnitRange{Int64}, label::Symbol, value}})
```

A AnnotatedString can also be created with `annotatedstring`, which acts much like `string` but preserves any annotations present in the arguments.

Examples

```
julia> AnnotatedString("this is an example annotated string",
                       [(1:18, :A, 1), (12:28, :B, 2), (18:35, :C, 3)])
"this is an example annotated string"
```

source

`Base.AnnotatedChar` – Type.

```
AnnotatedChar{S <: AbstractChar} <: AbstractChar
```

A Char with annotations.

More specifically, this is a simple wrapper around any other `AbstractChar`, which holds a list of arbitrary labelled annotations (`@NamedTuple{label::Symbol, value}`) with the wrapped character.

See also: `AnnotatedString`, `annotatedstring`, `annotations`, and `annotate!`.

Constructors

```
AnnotatedChar(s::S) -> AnnotatedChar{S}
AnnotatedChar(s::S, annotations::Vector{@NamedTuple{label::Symbol, value}})
```

Examples

```
julia> AnnotatedChar('j', [(:label, 1)])
'j': ASCII/Unicode U+006A (category Ll: Letter, lowercase)
```

source

Base.annotatedstring – Function.

```
annotatedstring(values...)
```

Create a AnnotatedString from any number of values using their printed representation.

This acts like string, but takes care to preserve any annotations present (in the form of AnnotatedString or AnnotatedChar values).

See also AnnotatedString and AnnotatedChar.

Examples

```
julia> annotatedstring("now an AnnotatedString")
"now an AnnotatedString"

julia> annotatedstring(AnnotatedString("annotated", [(1:9, :label, 1)]), ", and unannotated")
"annotated, and unannotated"
```

source

Base.annotations – Function.

```
annotations(chr::AnnotatedChar)::Vector{@NamedTuple{label::Symbol, value}}
```

Get all annotations of chr, in the form of a vector of annotation pairs.

source

```
annotations(str::Union{AnnotatedString, SubString{AnnotatedString}},
            [position::Union{Integer, UnitRange}]) ->
    Vector{@NamedTuple{region::UnitRange{Int64}, label::Symbol, value}}
```

Get all annotations that apply to str. Should position be provided, only annotations that overlap with position will be returned.

Annotations are provided together with the regions they apply to, in the form of a vector of region–annotation tuples.

In accordance with the semantics documented in AnnotatedString, the order of annotations returned matches the order in which they were applied.

See also: annotate!.

source

Base.annotate! – Function.

```
annotate!(char::AnnotatedChar, label::Symbol, value::Any)
```

Annotate char with the labeled value (label, value).

source

```
annotate!(str::AnnotatedString, [range::UnitRange{Int}], label::Symbol, value)
annotate!(str::SubString{AnnotatedString}, [range::UnitRange{Int}], label::Symbol, value)
```

Annotate a range of `str` (or the entire string) with a labeled value `(label, value)`. To remove existing `label` annotations, use a value of `nothing`.

The order in which annotations are applied to `str` is semantically meaningful, as described in `AnnotatedString`.

source

Chapter 49

Arrays

49.1 Constructors and Types

`Core.AbstractArray` – Type.

```
AbstractArray{T,N}
```

Supertype for N-dimensional arrays (or array-like types) with elements of type T. `Array` and other types are subtypes of this. See the manual section on the `AbstractArray` interface.

See also: `AbstractVector`, `AbstractMatrix`, `eltype`, `ndims`.

source

`Base.AbstractVector` – Type.

```
AbstractVector{T}
```

Supertype for one-dimensional arrays (or array-like types) with elements of type T. Alias for `AbstractArray{T,1}`.

source

`Base.AbstractMatrix` – Type.

```
AbstractMatrix{T}
```

Supertype for two-dimensional arrays (or array-like types) with elements of type T. Alias for `AbstractArray{T,2}`.

source

`Base.AbstractVecOrMat` – Type.

```
AbstractVecOrMat{T}
```

Union type of `AbstractVector{T}` and `AbstractMatrix{T}`.

source

`Core.Array` – Type.

```
Array{T,N} <: AbstractArray{T,N}
```

N-dimensional dense array with elements of type T.

source

Core.Array – Method.

```
Array{T}(undef, dims)
Array{T,N}(undef, dims)
```

Construct an uninitialized N-dimensional Array containing elements of type T. N can either be supplied explicitly, as in Array{T,N}(undef, dims), or be determined by the length or number of dims. dims may be a tuple or a series of integer arguments corresponding to the lengths in each dimension. If the rank N is supplied explicitly, then it must match the length or number of dims. Here undef is the UndefInitializer.

Examples

```
julia> A = Array{Float64, 2}(undef, 2, 3) # N given explicitly
2×3 Matrix{Float64}:
 6.90198e-310  6.90198e-310  6.90198e-310
 6.90198e-310  6.90198e-310  0.0

julia> B = Array{Float64}(undef, 4) # N determined by the input
4-element Vector{Float64}:
   2.360075077e-314
 NaN
   2.2671131793e-314
   2.299821756e-314

julia> similar(B, 2, 4, 1) # use typeof(B), and the given size
2×4×1 Array{Float64, 3}:
[:, :, 1] =
 2.26703e-314  2.26708e-314  0.0           2.80997e-314
 0.0           2.26703e-314  2.26708e-314  0.0
```

source

Core.Array – Method.

```
Array{T}(nothing, dims)
Array{T,N}(nothing, dims)
```

Construct an N-dimensional Array containing elements of type T, initialized with nothing entries. Element type T must be able to hold these values, i.e. Nothing <: T.

Examples

```
julia> Array{Union{Nothing, String}}(nothing, 2)
2-element Vector{Union{Nothing, String}}:
 nothing
 nothing

julia> Array{Union{Nothing, Int}}(nothing, 2, 3)
2×3 Matrix{Union{Nothing, Int64}}:
 nothing  nothing  nothing
 nothing  nothing  nothing
```

source

Core.Array – Method.

```
Array{T}(missing, dims)
Array{T,N}(missing, dims)
```

Construct an N-dimensional Array containing elements of type T, initialized with missing entries. Element type T must be able to hold these values, i.e. Missing <: T.

Examples

```
julia> Array{Union{Missing, String}}(missing, 2)
2-element Vector{Union{Missing, String}}:
 missing
 missing

julia> Array{Union{Missing, Int}}(missing, 2, 3)
2×3 Matrix{Union{Missing, Int64}}:
 missing  missing  missing
 missing  missing  missing
```

source

Core.UndefInitializer – Type.

```
UndefInitializer
```

Singleton type used in array initialization, indicating the array-constructor-caller would like an uninitialized array. See also undef, an alias for UndefInitializer().

Examples

```
julia> Array{Float64, 1}(UndefInitializer(), 3)
3-element Vector{Float64}:
 2.2752528595e-314
 2.202942107e-314
 2.275252907e-314
```

source

Core.undef – Constant.

```
undef
```

Alias for UndefInitializer(), which constructs an instance of the singleton type UndefInitializer, used in array initialization to indicate the array-constructor-caller would like an uninitialized array.

See also: missing, similar.

Examples

```
julia> Array{Float64, 1}(undef, 3)
3-element Vector{Float64}:
 2.2752528595e-314
 2.202942107e-314
 2.275252907e-314
```

source

Base.Vector – Type.

```
Vector{T} <: AbstractVector{T}
```

One-dimensional dense array with elements of type T, often used to represent a mathematical vector. Alias for `Array{T,1}`.

See also `empty`, `similar` and `zero` for creating vectors.

source

Base.Vector – Method.

```
Vector{T}(undef, n)
```

Construct an uninitialized `Vector{T}` of length `n`.

Examples

```
julia> Vector{Float64}(undef, 3)
3-element Vector{Float64}:
 6.90966e-310
 6.90966e-310
 6.90966e-310
```

source

Base.Vector – Method.

```
Vector{T}(nothing, m)
```

Construct a `Vector{T}` of length `m`, initialized with `nothing` entries. Element type T must be able to hold these values, i.e. `Nothing <: T`.

Examples

```
julia> Vector{Union{Nothing, String}}(nothing, 2)
2-element Vector{Union{Nothing, String}}:
 nothing
 nothing
```

source

Base.Vector – Method.

```
Vector{T}(missing, m)
```

Construct a `Vector{T}` of length `m`, initialized with `missing` entries. Element type T must be able to hold these values, i.e. `Missing <: T`.

Examples

```
julia> Vector{Union{Missing, String}}(missing, 2)
2-element Vector{Union{Missing, String}}:
 missing
 missing
```

source

Base.Matrix – Type.

```
Matrix{T} <: AbstractMatrix{T}
```

Two-dimensional dense array with elements of type T, often used to represent a mathematical matrix. Alias for Array{T,2}.

See also fill, zeros, undef and similar for creating matrices.

source

Base.Matrix – Method.

```
Matrix{T}(undef, m, n)
```

Construct an uninitialized Matrix{T} of size m×n.

Examples

```
julia> Matrix{Float64}(undef, 2, 3)
2×3 Matrix{Float64}:
 2.36365e-314  2.28473e-314    5.0e-324
 2.26704e-314  2.26711e-314  NaN

julia> similar(ans, Int32, 2, 2)
2×2 Matrix{Int32}:
 490537216  1277177453
         1  1936748399
```

source

Base.Matrix – Method.

```
Matrix{T}(nothing, m, n)
```

Construct a Matrix{T} of size m×n, initialized with nothing entries. Element type T must be able to hold these values, i.e. Nothing <: T.

Examples

```
julia> Matrix{Union{Nothing, String}}(nothing, 2, 3)
2×3 Matrix{Union{Nothing, String}}:
 nothing  nothing  nothing
 nothing  nothing  nothing
```

source

Base.Matrix – Method.

```
Matrix{T}(missing, m, n)
```

Construct a `Matrix{T}` of size `m×n`, initialized with `missing` entries. Element type T must be able to hold these values, i.e. `Missing <: T`.

Examples

```
julia> Matrix{Union{Missing, String}}(missing, 2, 3)
2×3 Matrix{Union{Missing, String}}:
 missing  missing  missing
 missing  missing  missing
```

source

`Base.VecOrMat` – Type.

```
VecOrMat{T}
```

Union type of `Vector{T}` and `Matrix{T}` which allows functions to accept either a Matrix or a Vector.

Examples

```
julia> Vector{Float64} <: VecOrMat{Float64}
true

julia> Matrix{Float64} <: VecOrMat{Float64}
true

julia> Array{Float64, 3} <: VecOrMat{Float64}
false
```

source

`Core.DenseArray` – Type.

```
DenseArray{T, N} <: AbstractArray{T,N}
```

N-dimensional dense array with elements of type T. The elements of a dense array are stored contiguously in memory.

source

`Base.DenseVector` – Type.

```
DenseVector{T}
```

One-dimensional `DenseArray` with elements of type T. Alias for `DenseArray{T,1}`.

source

`Base.DenseMatrix` – Type.

```
DenseMatrix{T}
```

Two-dimensional `DenseArray` with elements of type T. Alias for `DenseArray{T,2}`.

source

Base.DenseVecOrMat – Type.

```
DenseVecOrMat{T}
```

Union type of DenseVector{T} and DenseMatrix{T}.

source

Base.StridedArray – Type.

```
StridedArray{T, N}
```

A hard-coded Union of common array types that follow the strided array interface, with elements of type T and N dimensions.

If A is a StridedArray, then its elements are stored in memory with offsets, which may vary between dimensions but are constant within a dimension. For example, A could have stride 2 in dimension 1, and stride 3 in dimension 2. Incrementing A along dimension d jumps in memory by [stride(A, d)] slots. Strided arrays are particularly important and useful because they can sometimes be passed directly as pointers to foreign language libraries like BLAS.

source

Base.StridedVector – Type.

```
StridedVector{T}
```

One dimensional StridedArray with elements of type T.

source

Base.StridedMatrix – Type.

```
StridedMatrix{T}
```

Two dimensional StridedArray with elements of type T.

source

Base.StridedVecOrMat – Type.

```
StridedVecOrMat{T}
```

Union type of StridedVector and StridedMatrix with elements of type T.

source

Core.GenericMemory – Type.

```
GenericMemory{kind::Symbol, T, addrspace=Core.CPU} <: DenseVector{T}
```

Fixed-size DenseVector{T}.

kind can currently be either :not_atomic or :atomic. For details on what :atomic implies, see AtomicMemory

addrspace can currently only be set to Core.CPU. It is designed to permit extension by other systems such as GPUs, which might define values such as:

```
module CUDA
const Generic = bitcast(Core.AddrSpace{CUDA}, 0)
const Global = bitcast(Core.AddrSpace{CUDA}, 1)
end
```

The exact semantics of these other addrspaces is defined by the specific backend, but will error if the user is attempting to access these on the CPU.

Julia 1.11

This type requires Julia 1.11 or later.

source

`Core.Memory` – Type.

```
Memory{T} == GenericMemory{:not_atomic, T, Core.CPU}
```

Fixed-size `DenseVector{T}`.

Julia 1.11

This type requires Julia 1.11 or later.

source

`Core.Memory` – Method.

```
Memory{T}(undef, n)
```

Construct an uninitialized `Memory{T}` of length n. All Memory objects of length 0 might alias, since there is no reachable mutable content from them.

Examples

```
julia> Memory{Float64}(undef, 3)
3-element Memory{Float64}:
 6.90966e-310
 6.90966e-310
 6.90966e-310
```

source

`Core.memoryref` – Function.

```
memoryref(::GenericMemory, index::Integer)
memoryref(::GenericMemoryRef, index::Integer)
```

Construct a `GenericMemoryRef` from a memory object and an offset index (1-based) which can also be negative. This always returns an inbounds object, and will throw an error if that is not possible (because the index would result in a shift out-of-bounds of the underlying memory).

source

```
memoryref(::GenericMemory)
```

Construct a GenericMemoryRef from a memory object. This does not fail, but the resulting memory will point out-of-bounds if and only if the memory is empty.

source

Base.memoryindex – Function.

```
memoryindex(ref::GenericMemoryRef)::Int
```

Get the 1-based index of ref in its GenericMemory.

Examples

```
julia> mem = Memory{String}(undef, 10);

julia> ref = Base.memoryindex(memoryref(mem, 3))
3

julia> Base.memoryindex(memoryref(Memory{Nothing}(undef, 10), 8))
8
```

Julia 1.13

This function requires at least Julia 1.13.

source

Base.Slices – Type.

```
Slices{P,SM,AX,S,N} <: AbstractSlices{S,N}
```

An AbstractArray of slices into a parent array over specified dimension(s), returning views that select all the data from the other dimension(s).

These should typically be constructed by eachslice, eachcol or eachrow.

parent(s::Slices) will return the parent array.

source

Base.RowSlices – Type.

```
RowSlices{M,AX,S}
```

A special case of Slices that is a vector of row slices of a matrix, as constructed by eachrow.

parent can be used to get the underlying matrix.

source

Base.ColumnSlices – Type.

```
ColumnSlices{M,AX,S}
```

A special case of Slices that is a vector of column slices of a matrix, as constructed by eachcol.

parent can be used to get the underlying matrix.

source

Base.getindex – Method.

```
getindex(type[, elements...])
```

Construct a 1-d array of the specified type. This is usually called with the syntax Type[]. Element values can be specified using Type[a,b,c,...].

Examples

```
julia> Int8[1, 2, 3]
3-element Vector{Int8}:
 1
 2
 3

julia> getindex(Int8, 1, 2, 3)
3-element Vector{Int8}:
 1
 2
 3
```

source

Base.zeros – Function.

```
zeros([T=Float64,] dims::Tuple)
zeros([T=Float64,] dims...)
```

Create an Array, with element type T, of all zeros with size specified by dims. See also fill, ones, zero.

Examples

```
julia> zeros(1)
1-element Vector{Float64}:
 0.0

julia> zeros(Int8, 2, 3)
2×3 Matrix{Int8}:
 0  0  0
 0  0  0
```

source

Base.ones – Function.

```
ones([T=Float64,] dims::Tuple)
ones([T=Float64,] dims...)
```

Create an Array, with element type T, of all ones with size specified by dims. See also fill, zeros.

Examples

```
julia> ones(1,2)
1×2 Matrix{Float64}:
 1.0  1.0

julia> ones(ComplexF64, 2, 3)
2×3 Matrix{ComplexF64}:
 1.0+0.0im  1.0+0.0im  1.0+0.0im
 1.0+0.0im  1.0+0.0im  1.0+0.0im
```

source

Base.BitArray – Type.

```
BitArray{N} <: AbstractArray{Bool, N}
```

Space-efficient N-dimensional boolean array, using just one bit for each boolean value.

BitArrays pack up to 64 values into every 8 bytes, resulting in an 8x space efficiency over Array{Bool, N} and allowing some operations to work on 64 values at once.

By default, Julia returns BitArrays from broadcasting operations that generate boolean elements (including dotted-comparisons like .==) as well as from the functions trues and falses.

Note

Due to its packed storage format, concurrent access to the elements of a BitArray where at least one of them is a write is not thread-safe.

source

Base.BitArray – Method.

```
BitArray(undef, dims::Integer...)
BitArray{N}(undef, dims::NTuple{N,Int})
```

Construct an undef BitArray with the given dimensions. Behaves identically to the Array constructor. See undef.

Examples

```
julia> BitArray(undef, 2, 2)
2×2 BitMatrix:
 0  0
 0  0

julia> BitArray(undef, (3, 1))
3×1 BitMatrix:
 0
 0
 0
```

source

Base.BitArray – Method.

```
BitArray(itr)
```

Construct a `BitArray` generated by the given iterable object. The shape is inferred from the `itr` object.

Examples

```
julia> BitArray([1 0; 0 1])
2×2 BitMatrix:
 1  0
 0  1

julia> BitArray(x+y == 3 for x = 1:2, y = 1:3)
2×3 BitMatrix:
 0  1  0
 1  0  0

julia> BitArray(x+y == 3 for x = 1:2 for y = 1:3)
6-element BitVector:
 0
 1
 0
 1
 0
 0
```

source

Base.trues – Function.

```
trues(dims)
```

Create a `BitArray` with all values set to `true`.

Examples

```
julia> trues(2,3)
2×3 BitMatrix:
 1  1  1
 1  1  1
```

source

Base.falses – Function.

```
falses(dims)
```

Create a `BitArray` with all values set to `false`.

Examples

```
julia> falses(2,3)
2×3 BitMatrix:
 0  0  0
 0  0  0
```

source

Base.fill – Function.

```
fill(value, dims::Tuple)
fill(value, dims...)
```

Create an array of size dims with every location set to value.

For example, fill(1.0, (5,5)) returns a 5×5 array of floats, with 1.0 in every location of the array.

The dimension lengths dims may be specified as either a tuple or a sequence of arguments. An N-length tuple or N arguments following the value specify an N-dimensional array. Thus, a common idiom for creating a zero-dimensional array with its only location set to x is fill(x).

Every location of the returned array is set to (and is thus === to) the value that was passed; this means that if the value is itself modified, all elements of the filled array will reflect that modification because they're *still* that very value. This is of no concern with fill(1.0, (5,5)) as the value 1.0 is immutable and cannot itself be modified, but can be unexpected with mutable values like — most commonly — arrays. For example, fill([], 3) places *the very same* empty array in all three locations of the returned vector:

```
julia> v = fill([], 3)
3-element Vector{Vector{Any}}:
 []
 []
 []

julia> v[1] === v[2] === v[3]
true

julia> value = v[1]
Any[]

julia> push!(value, 867_5309)
1-element Vector{Any}:
 8675309

julia> v
3-element Vector{Vector{Any}}:
 [8675309]
 [8675309]
 [8675309]
```

To create an array of many independent inner arrays, use a comprehension instead. This creates a new and distinct array on each iteration of the loop:

```
julia> v2 = [[] for _ in 1:3]
3-element Vector{Vector{Any}}:
 []
 []
 []

julia> v2[1] === v2[2] === v2[3]
```

```
false

julia> push!(v2[1], 8675309)
1-element Vector{Any}:
 8675309

julia> v2
3-element Vector{Vector{Any}}:
 [8675309]
 []
 []
```

See also: `fill!`, `zeros`, `ones`, `similar`.

Examples

```
julia> fill(1.0, (2,3))
2×3 Matrix{Float64}:
 1.0  1.0  1.0
 1.0  1.0  1.0

julia> fill(42)
0-dimensional Array{Int64, 0}:
42

julia> A = fill(zeros(2), 2) # sets both elements to the same [0.0, 0.0] vector
2-element Vector{Vector{Float64}}:
 [0.0, 0.0]
 [0.0, 0.0]

julia> A[1][1] = 42; # modifies the filled value to be [42.0, 0.0]

julia> A # both A[1] and A[2] are the very same vector
2-element Vector{Vector{Float64}}:
 [42.0, 0.0]
 [42.0, 0.0]
```

source

`Base.fill!` – Function.

```
fill!(A, x)
```

Fill array `A` with the value `x`. If `x` is an object reference, all elements will refer to the same object. `fill!(A, Foo())` will return `A` filled with the result of evaluating `Foo()` once.

Examples

```
julia> A = zeros(2,3)
2×3 Matrix{Float64}:
 0.0  0.0  0.0
 0.0  0.0  0.0

julia> fill!(A, 2.)
```

```
2×3 Matrix{Float64}:
 2.0  2.0  2.0
 2.0  2.0  2.0

julia> a = [1, 1, 1]; A = fill!(Vector{Vector{Int}}(undef, 3), a); a[1] = 2; A
3-element Vector{Vector{Int64}}:
 [2, 1, 1]
 [2, 1, 1]
 [2, 1, 1]

julia> x = 0; f() = (global x += 1; x); fill!(Vector{Int}(undef, 3), f())
3-element Vector{Int64}:
 1
 1
 1
```

source

Base.empty – Function.

```
empty(a::AbstractDict, [index_type=keytype(a)], [value_type=valtype(a)])
```

Create an empty `AbstractDict` container which can accept indices of type `index_type` and values of type `value_type`. The second and third arguments are optional and default to the input's `keytype` and `valtype`, respectively. (If only one of the two types is specified, it is assumed to be the `value_type`, and the `index_type` we default to `keytype(a)`).

Custom `AbstractDict` subtypes may choose which specific dictionary type is best suited to return for the given index and value types, by specializing on the three-argument signature. The default is to return an empty `Dict`.

source

```
empty(v::AbstractVector, [eltype])
```

Create an empty vector similar to `v`, optionally changing the `eltype`.

See also: `empty!`, `isempty`, `isassigned`.

Examples

```
julia> empty([1.0, 2.0, 3.0])
Float64[]

julia> empty([1.0, 2.0, 3.0], String)
String[]
```

source

```
empty(x::Tuple)
```

Return an empty tuple, `()`.

source

Base.similar – Function.

```
similar(storagetype, axes)
```

Create an uninitialized mutable array analogous to that specified by storagetype, but with axes specified by the last argument.

Examples:

```
similar(Array{Int}, axes(A))
```

creates an array that "acts like" an Array{Int} (and might indeed be backed by one), but which is indexed identically to A. If A has conventional indexing, this will be identical to Array{Int}(undef, size(A)), but if A has unconventional indexing then the indices of the result will match A.

```
similar(BitArray, (axes(A, 2),))
```

would create a 1-dimensional logical array whose indices match those of the columns of A.

source

```
similar(array, [element_type=eltype(array)], [dims=size(array)])
```

Create an uninitialized mutable array with the given element type and size, based upon the given source array. The second and third arguments are both optional, defaulting to the given array's eltype and size. The dimensions may be specified either as a single tuple argument or as a series of integer arguments.

Custom AbstractArray subtypes may choose which specific array type is best-suited to return for the given element type and dimensionality. If they do not specialize this method, the default is an Array{element_type}(undef, dims...).

For example, similar(1:10, 1, 4) returns an uninitialized Array{Int,2} since ranges are neither mutable nor support 2 dimensions:

```
julia> similar(1:10, 1, 4)
1×4 Matrix{Int64}:
 4419743872  4374413872  4419743888  0
```

Conversely, similar(trues(10,10), 2) returns an uninitialized BitVector with two elements since BitArrays are both mutable and can support 1-dimensional arrays:

```
julia> similar(trues(10,10), 2)
2-element BitVector:
 0
 0
```

Since BitArrays can only store elements of type Bool, however, if you request a different element type it will create a regular Array instead:

```
julia> similar(falses(10), Float64, 2, 4)
2×4 Matrix{Float64}:
 2.18425e-314  2.18425e-314  2.18425e-314  2.18425e-314
 2.18425e-314  2.18425e-314  2.18425e-314  2.18425e-314
```

See also: undef, isassigned.

source

```
similar(A::AbstractSparseMatrixCSC{Tv,Ti}, [::Type{TvNew}, ::Type{TiNew}, m::Integer,
↪  n::Integer]) where {Tv,Ti}
```

Create an uninitialized mutable array with the given element type, index type, and size, based upon the given source SparseMatrixCSC. The new sparse matrix maintains the structure of the original sparse matrix, except in the case where dimensions of the output matrix are different from the output.

The output matrix has zeros in the same locations as the input, but uninitialized values for the nonzero locations.

49.2 Basic functions

Base.ndims – Function.

```
ndims(A::AbstractArray)::Integer
```

Return the number of dimensions of A.

See also: size, axes.

Examples

```
julia> A = fill(1, (3,4,5));

julia> ndims(A)
3
```

source

Base.size – Function.

```
size(A::AbstractArray, [dim])
```

Return a tuple containing the dimensions of A. Optionally you can specify a dimension to just get the length of that dimension.

Note that size may not be defined for arrays with non-standard indices, in which case axes may be useful. See the manual chapter on arrays with custom indices.

See also: length, ndims, eachindex, sizeof.

Examples

```
julia> A = fill(1, (2,3,4));

julia> size(A)
(2, 3, 4)

julia> size(A, 2)
3
```

source

Base.axes – Method.

```
axes(A)
```

Return the tuple of valid indices for array A.

See also: size, keys, eachindex.

Examples

```
julia> A = fill(1, (5,6,7));

julia> axes(A)
(Base.OneTo(5), Base.OneTo(6), Base.OneTo(7))
```

source

Base.axes – Method.

```
axes(A, d)
```

Return the valid range of indices for array A along dimension d.

See also size, and the manual chapter on arrays with custom indices.

Examples

```
julia> A = fill(1, (5,6,7));

julia> axes(A, 2)
Base.OneTo(6)

julia> axes(A, 4) == 1:1  # all dimensions d > ndims(A) have size 1
true
```

Usage note

Each of the indices has to be an AbstractUnitRange{<:Integer}, but at the same time can be a type that uses custom indices. So, for example, if you need a subset, use generalized indexing constructs like begin/end or firstindex/lastindex:

```
ix = axes(v, 1)
ix[2:end]          # will work for eg Vector, but may fail in general
ix[(begin+1):end]  # works for generalized indexes
```

source

Base.length – Method.

```
length(A::AbstractArray)
```

Return the number of elements in the array, defaults to prod(size(A)).

Examples

```
julia> length([1, 2, 3, 4])
4

julia> length([1 2; 3 4])
4
```

source

Base.keys – Method.

```
keys(a::AbstractArray)
```

Return an efficient array describing all valid indices for a arranged in the shape of a itself.

The keys of 1-dimensional arrays (vectors) are integers, whereas all other N-dimensional arrays use CartesianIndex to describe their locations. Often the special array types LinearIndices and CartesianIndices are used to efficiently represent these arrays of integers and CartesianIndexes, respectively.

Note that the keys of an array might not be the most efficient index type; for maximum performance use eachindex instead.

Examples

```
julia> keys([4, 5, 6])
3-element LinearIndices{1, Tuple{Base.OneTo{Int64}}}:
 1
 2
 3

julia> keys([4 5; 6 7])
CartesianIndices((2, 2))
```

source

Base.eachindex – Function.

```
eachindex(A...)
eachindex(::IndexStyle, A::AbstractArray...)
```

Create an iterable object for visiting each index of an AbstractArray A in an efficient manner. For array types that have opted into fast linear indexing (like Array), this is simply the range 1:length(A) if they use 1-based indexing. For array types that have not opted into fast linear indexing, a specialized Cartesian range is typically returned to efficiently index into the array with indices specified for every dimension.

In general eachindex accepts arbitrary iterables, including strings and dictionaries, and returns an iterator object supporting arbitrary index types (e.g. unevenly spaced or non-integer indices).

If A is AbstractArray it is possible to explicitly specify the style of the indices that should be returned by eachindex by passing a value having IndexStyle type as its first argument (typically IndexLinear() if linear indices are required or IndexCartesian() if Cartesian range is wanted).

If you supply more than one AbstractArray argument, eachindex will create an iterable object that is fast for all arguments (typically a UnitRange if all inputs have fast linear indexing, a CartesianIndices

otherwise). If the arrays have different sizes and/or dimensionalities, a `DimensionMismatch` exception will be thrown.

See also `pairs(A)` to iterate over indices and values together, and `axes(A, 2)` for valid indices along one dimension.

Examples

```
julia> A = [10 20; 30 40];

julia> for i in eachindex(A) # linear indexing
           println("A[", i, "] == ", A[i])
       end
A[1] == 10
A[2] == 30
A[3] == 20
A[4] == 40

julia> for i in eachindex(view(A, 1:2, 1:1)) # Cartesian indexing
           println(i)
       end
CartesianIndex(1, 1)
CartesianIndex(2, 1)
```

source

`Base.IndexStyle` – Type.

```
IndexStyle(A)
IndexStyle(typeof(A))
```

`IndexStyle` specifies the "native indexing style" for array `A`. When you define a new `AbstractArray` type, you can choose to implement either linear indexing (with `IndexLinear`) or cartesian indexing. If you decide to only implement linear indexing, then you must set this trait for your array type:

```
Base.IndexStyle(::Type{<:MyArray}) = IndexLinear()
```

The default is `IndexCartesian()`.

Julia's internal indexing machinery will automatically (and invisibly) recompute all indexing operations into the preferred style. This allows users to access elements of your array using any indexing style, even when explicit methods have not been provided.

If you define both styles of indexing for your `AbstractArray`, this trait can be used to select the most performant indexing style. Some methods check this trait on their inputs, and dispatch to different algorithms depending on the most efficient access pattern. In particular, `eachindex` creates an iterator whose type depends on the setting of this trait.

source

`Base.IndexLinear` – Type.

```
IndexLinear()
```

Subtype of `IndexStyle` used to describe arrays which are optimally indexed by one linear index.

A linear indexing style uses one integer index to describe the position in the array (even if it's a multidimensional array) and column-major ordering is used to efficiently access the elements. This means that requesting `eachindex` from an array that is `IndexLinear` will return a simple one-dimensional range, even if it is multidimensional.

A custom array that reports its `IndexStyle` as `IndexLinear` only needs to implement indexing (and indexed assignment) with a single `Int` index; all other indexing expressions — including multidimensional accesses — will be recomputed to the linear index. For example, if `A` were a 2×3 custom matrix with linear indexing, and we referenced `A[1, 3]`, this would be recomputed to the equivalent linear index and call `A[5]` since `1 + 2*(3 - 1) = 5`.

See also `IndexCartesian`.

source

`Base.IndexCartesian` – Type.

```
IndexCartesian()
```

Subtype of `IndexStyle` used to describe arrays which are optimally indexed by a Cartesian index. This is the default for new custom `AbstractArray` subtypes.

A Cartesian indexing style uses multiple integer indices to describe the position in a multidimensional array, with exactly one index per dimension. This means that requesting `eachindex` from an array that is `IndexCartesian` will return a range of `CartesianIndices`.

A `N`-dimensional custom array that reports its `IndexStyle` as `IndexCartesian` needs to implement indexing (and indexed assignment) with exactly `N` `Int` indices; all other indexing expressions — including linear indexing — will be recomputed to the equivalent Cartesian location. For example, if `A` were a 2×3 custom matrix with cartesian indexing, and we referenced `A[5]`, this would be recomputed to the equivalent Cartesian index and call `A[1, 3]` since `5 = 1 + 2*(3 - 1)`.

It is significantly more expensive to compute Cartesian indices from a linear index than it is to go the other way. The former operation requires division — a very costly operation — whereas the latter only uses multiplication and addition and is essentially free. This asymmetry means it is far more costly to use linear indexing with an `IndexCartesian` array than it is to use Cartesian indexing with an `IndexLinear` array.

See also `IndexLinear`.

source

`Base.conj!` – Function.

```
conj!(A)
```

Transform an array to its complex conjugate in-place.

See also `conj`.

Examples

```
julia> A = [1+im 2-im; 2+2im 3+im]
2×2 Matrix{Complex{Int64}}:
 1+1im  2-1im
```

```
 2+2im  3+1im

julia> conj!(A);

julia> A
2×2 Matrix{Complex{Int64}}:
 1-1im  2+1im
 2-2im  3-1im
```

source

Base.stride – Function.

```
stride(A, k::Integer)
```

Return the distance in memory (in number of elements) between adjacent elements in dimension k.

See also: strides.

Examples

```
julia> A = fill(1, (3,4,5));

julia> stride(A,2)
3

julia> stride(A,3)
12
```

source

Base.strides – Function.

```
strides(A)
```

Return a tuple of the memory strides in each dimension.

See also: stride.

Examples

```
julia> A = fill(1, (3,4,5));

julia> strides(A)
(1, 3, 12)
```

source

49.3 Broadcast and vectorization

See also the dot syntax for vectorizing functions; for example, `f.(args...)` implicitly calls `broadcast(f, args...)`. Rather than relying on "vectorized" methods of functions like `sin` to operate on arrays, you should use `sin.(a)` to vectorize via `broadcast`.

Base.Broadcast.broadcast – Function.

```
broadcast(f, As...)
```

Broadcast the function `f` over the arrays, tuples, collections, `Ref`s and/or scalars `As`.

Broadcasting applies the function `f` over the elements of the container arguments and the scalars themselves in `As`. Singleton and missing dimensions are expanded to match the extents of the other arguments by virtually repeating the value. By default, only a limited number of types are considered scalars, including `Number`s, `String`s, `Symbol`s, `Type`s, `Function`s and some common singletons like `missing` and `nothing`. All other arguments are iterated over or indexed into elementwise.

The resulting container type is established by the following rules:

- If all the arguments are scalars or zero-dimensional arrays, it returns an unwrapped scalar.
- If at least one argument is a tuple and all others are scalars or zero-dimensional arrays, it returns a tuple.
- All other combinations of arguments default to returning an `Array`, but custom container types can define their own implementation and promotion-like rules to customize the result when they appear as arguments.
- The element type is determined in the same manner as in `collect`.

A special syntax exists for broadcasting: `f.(args...)` is equivalent to `broadcast(f, args...)`, and nested `f.(g.(args...))` calls are fused into a single broadcast loop.

Examples

```
julia> A = [1, 2, 3, 4, 5]
5-element Vector{Int64}:
 1
 2
 3
 4
 5

julia> B = [1 2; 3 4; 5 6; 7 8; 9 10]
5×2 Matrix{Int64}:
 1   2
 3   4
 5   6
 7   8
 9  10

julia> broadcast(+, A, B)
5×2 Matrix{Int64}:
  2   3
  5   6
  8   9
 11  12
 14  15

julia> parse.(Int, ["1", "2"])
2-element Vector{Int64}:
 1
```

```
 2

julia> abs.((1, -2))
(1, 2)

julia> broadcast(+, 1.0, (0, -2.0))
(1.0, -1.0)

julia> (+).([[0,2], [1,3]], Ref{Vector{Int}}([1,-1]))
2-element Vector{Vector{Int64}}:
 [1, 1]
 [2, 2]

julia> string.(("one","two","three","four"), ": ", 1:4)
4-element Vector{String}:
 "one: 1"
 "two: 2"
 "three: 3"
 "four: 4"
```

source

Base.Broadcast.broadcast! – Function.

```
broadcast!(f, dest, As...)
```

Like broadcast, but store the result of broadcast(f, As...) in the dest array. Note that dest is only used to store the result, and does not supply arguments to f unless it is also listed in the As, as in broadcast!(f, A, A, B) to perform A[:] = broadcast(f, A, B).

Examples

```
julia> A = [1.0; 0.0]; B = [0.0; 0.0];

julia> broadcast!(+, B, A, (0, -2.0));

julia> B
2-element Vector{Float64}:
  1.0
 -2.0

julia> A
2-element Vector{Float64}:
 1.0
 0.0

julia> broadcast!(+, A, A, (0, -2.0));

julia> A
2-element Vector{Float64}:
  1.0
 -2.0
```

source

Base.Broadcast.@__dot__ – Macro.

```
@. expr
```

Convert every function call or operator in expr into a "dot call" (e.g. convert f(x) to f.(x)), and convert every assignment in expr to a "dot assignment" (e.g. convert += to .+=).

If you want to *avoid* adding dots for selected function calls in expr, splice those function calls in with $. For example, @. sqrt(abs($sort(x))) is equivalent to sqrt.(abs.(sort(x))) (no dot for sort).

(@. is equivalent to a call to @__dot__.)

Examples

```
julia> x = 1.0:3.0; y = similar(x);

julia> @. y = x + 3 * sin(x)
3-element Vector{Float64}:
 3.5244129544236893
 4.727892280477045
 3.4233600241796016
```

source

Base.Broadcast.BroadcastFunction – Type.

```
BroadcastFunction{F} <: Function
```

Represents the "dotted" version of an operator, which broadcasts the operator over its arguments, so BroadcastFunction(op) is functionally equivalent to (x...) -> (op).(x...).

Can be created by just passing an operator preceded by a dot to a higher-order function.

Examples

```
julia> a = [[1 3; 2 4], [5 7; 6 8]];

julia> b = [[9 11; 10 12], [13 15; 14 16]];

julia> map(.*, a, b)
2-element Vector{Matrix{Int64}}:
 [9 33; 20 48]
 [65 105; 84 128]

julia> Base.BroadcastFunction(+)(a, b) == a .+ b
true
```

Julia 1.6

BroadcastFunction and the standalone .op syntax are available as of Julia 1.6.

source

For specializing broadcast on custom types, see

`Base.Broadcast.BroadcastStyle` – Type.

> `BroadcastStyle` is an abstract type and trait-function used to determine behavior of objects under broadcasting. `BroadcastStyle(typeof(x))` returns the style associated with `x`. To customize the broadcasting behavior of a type, one can declare a style by defining a type/method pair
>
> ```
> struct MyContainerStyle <: BroadcastStyle end
> Base.BroadcastStyle(::Type{<:MyContainer}) = MyContainerStyle()
> ```
>
> One then writes method(s) (at least `similar`) operating on `Broadcasted{MyContainerStyle}`. There are also several pre-defined subtypes of `BroadcastStyle` that you may be able to leverage; see the Interfaces chapter for more information.
>
> source

`Base.Broadcast.AbstractArrayStyle` – Type.

> `Broadcast.AbstractArrayStyle{N} <: BroadcastStyle` is the abstract supertype for any style associated with an `AbstractArray` type. The `N` parameter is the dimensionality, which can be handy for AbstractArray types that only support specific dimensionalities:
>
> ```
> struct SparseMatrixStyle <: Broadcast.AbstractArrayStyle{2} end
> Base.BroadcastStyle(::Type{<:SparseMatrixCSC}) = SparseMatrixStyle()
> ```
>
> For `AbstractArray` types that support arbitrary dimensionality, `N` can be set to `Any`:
>
> ```
> struct MyArrayStyle <: Broadcast.AbstractArrayStyle{Any} end
> Base.BroadcastStyle(::Type{<:MyArray}) = MyArrayStyle()
> ```
>
> In cases where you want to be able to mix multiple `AbstractArrayStyles` and keep track of dimensionality, your style needs to support a `Val` constructor:
>
> ```
> struct MyArrayStyleDim{N} <: Broadcast.AbstractArrayStyle{N} end
> (::Type{<:MyArrayStyleDim})(::Val{N}) where N = MyArrayStyleDim{N}()
> ```
>
> Note that if two or more `AbstractArrayStyle` subtypes conflict, broadcasting machinery will fall back to producing `Arrays`. If this is undesirable, you may need to define binary `BroadcastStyle` rules to control the output type.
>
> See also `Broadcast.DefaultArrayStyle`.
>
> source

`Base.Broadcast.ArrayStyle` – Type.

> `Broadcast.ArrayStyle{MyArrayType}()` is a `BroadcastStyle` indicating that an object behaves as an array for broadcasting. It presents a simple way to construct `Broadcast.AbstractArrayStyle`s for specific `AbstractArray` container types. Broadcast styles created this way lose track of dimensionality; if keeping track is important for your type, you should create your own custom `Broadcast.AbstractArrayStyle`.
>
> source

`Base.Broadcast.DefaultArrayStyle` – Type.

`Broadcast.DefaultArrayStyle{N}()` is a `BroadcastStyle` indicating that an object behaves as an `N`-dimensional array for broadcasting. Specifically, `DefaultArrayStyle` is used for any `AbstractArray` type that hasn't defined a specialized style, and in the absence of overrides from other `broadcast` arguments the resulting output type is `Array`. When there are multiple inputs to `broadcast`, `DefaultArrayStyle` "loses" to any other `Broadcast.ArrayStyle`.

source

`Base.Broadcast.broadcastable` – Function.

```
Broadcast.broadcastable(x)
```

Return either `x` or an object like `x` such that it supports `axes`, indexing, and its type supports `ndims`.

If `x` supports iteration, the returned value should have the same axes and indexing behaviors as `collect(x)`.

If `x` is not an `AbstractArray` but it supports `axes`, indexing, and its type supports `ndims`, then `broadcastable(::typeof(x))` may be implemented to just return itself. Further, if `x` defines its own `BroadcastStyle`, then it must define its `broadcastable` method to return itself for the custom style to have any effect.

Examples

```
julia> Broadcast.broadcastable([1,2,3]) # like `identity` since arrays already support axes
↪  and indexing
3-element Vector{Int64}:
 1
 2
 3

julia> Broadcast.broadcastable(Int) # Types don't support axes, indexing, or iteration but are
↪  commonly used as scalars
Base.RefValue{Type{Int64}}(Int64)

julia> Broadcast.broadcastable("hello") # Strings break convention of matching iteration and
↪  act like a scalar instead
Base.RefValue{String}("hello")
```

source

`Base.Broadcast.combine_axes` – Function.

```
combine_axes(As...)::Tuple
```

Determine the result axes for broadcasting across all values in `As`.

```
julia> Broadcast.combine_axes([1], [1 2; 3 4; 5 6])
(Base.OneTo(3), Base.OneTo(2))

julia> Broadcast.combine_axes(1, 1, 1)
()
```

source

`Base.Broadcast.combine_styles` – Function.

```
combine_styles(cs...)::BroadcastStyle
```

Decides which `BroadcastStyle` to use for any number of value arguments. Uses `BroadcastStyle` to get the style for each argument, and uses `result_style` to combine styles.

Examples

```
julia> Broadcast.combine_styles([1], [1 2; 3 4])
Base.Broadcast.DefaultArrayStyle{2}()
```

source

`Base.Broadcast.result_style` – Function.

```
result_style(s1::BroadcastStyle[, s2::BroadcastStyle])::BroadcastStyle
```

Takes one or two `BroadcastStyles` and combines them using `BroadcastStyle` to determine a common `BroadcastStyle`.

Examples

```
julia> Broadcast.result_style(Broadcast.DefaultArrayStyle{0}(),
↪  Broadcast.DefaultArrayStyle{3}())
Base.Broadcast.DefaultArrayStyle{3}()

julia> Broadcast.result_style(Broadcast.Unknown(), Broadcast.DefaultArrayStyle{1}())
Base.Broadcast.DefaultArrayStyle{1}()
```

source

49.4 Indexing and assignment

`Base.getindex` – Method.

```
getindex(A, inds...)
```

Return a subset of array `A` as selected by the indices `inds`.

Each index may be any supported index type, such as an `Integer`, `CartesianIndex`, range, or array of supported indices. A `:` may be used to select all elements along a specific dimension, and a boolean array (e.g. an `Array{Bool}` or a `BitArray`) may be used to filter for elements where the corresponding index is `true`.

When `inds` selects multiple elements, this function returns a newly allocated array. To index multiple elements without making a copy, use `view` instead.

See the manual section on array indexing for details.

Examples

```
julia> A = [1 2; 3 4]
2×2 Matrix{Int64}:
 1  2
 3  4
```

```
julia> getindex(A, 1)
1

julia> getindex(A, [2, 1])
2-element Vector{Int64}:
 3
 1

julia> getindex(A, 2:4)
3-element Vector{Int64}:
 3
 2
 4

julia> getindex(A, 2, 1)
3

julia> getindex(A, CartesianIndex(2, 1))
3

julia> getindex(A, :, 2)
2-element Vector{Int64}:
 2
 4

julia> getindex(A, 2, :)
2-element Vector{Int64}:
 3
 4

julia> getindex(A, A .> 2)
2-element Vector{Int64}:
 3
 4
```

source

Base.setindex! – Method.

```
setindex!(A, X, inds...)
A[inds...] = X
```

Store values from array X within some subset of A as specified by inds. The syntax A[inds...] = X is equivalent to (setindex!(A, X, inds...); X).

Warning

Behavior can be unexpected when any mutated argument shares memory with any other argument.

Examples

```
julia> A = zeros(2,2);
```

```
julia> setindex!(A, [10, 20], [1, 2]);

julia> A[[3, 4]] = [30, 40];

julia> A
2×2 Matrix{Float64}:
 10.0  30.0
 20.0  40.0
```

source

Base.nextind – Function.

```
nextind(A, i)
```

Return the index after `i` in `A`. The returned index is often equivalent to `i + 1` for an integer `i`. This function can be useful for generic code.

Warning

The returned index might be out of bounds. Consider using `checkbounds`.

See also: `prevind`.

Examples

```
julia> x = [1 2; 3 4]
2×2 Matrix{Int64}:
 1  2
 3  4

julia> nextind(x, 1) # valid result
2

julia> nextind(x, 4) # invalid result
5

julia> nextind(x, CartesianIndex(1, 1)) # valid result
CartesianIndex(2, 1)

julia> nextind(x, CartesianIndex(2, 2)) # invalid result
CartesianIndex(1, 3)
```

source

Base.prevind – Function.

```
prevind(A, i)
```

Return the index before `i` in `A`. The returned index is often equivalent to `i - 1` for an integer `i`. This function can be useful for generic code.

Warning

The returned index might be out of bounds. Consider using `checkbounds`.

See also: `nextind`.

Examples

```
julia> x = [1 2; 3 4]
2×2 Matrix{Int64}:
 1  2
 3  4

julia> prevind(x, 4) # valid result
3

julia> prevind(x, 1) # invalid result
0

julia> prevind(x, CartesianIndex(2, 2)) # valid result
CartesianIndex(1, 2)

julia> prevind(x, CartesianIndex(1, 1)) # invalid result
CartesianIndex(2, 0)
```

source

`Base.copyto!` – Method.

```
copyto!(dest, Rdest::CartesianIndices, src, Rsrc::CartesianIndices) -> dest
```

Copy the block of `src` in the range of `Rsrc` to the block of `dest` in the range of `Rdest`. The sizes of the two regions must match.

Examples

```
julia> A = zeros(5, 5);

julia> B = [1 2; 3 4];

julia> Ainds = CartesianIndices((2:3, 2:3));

julia> Binds = CartesianIndices(B);

julia> copyto!(A, Ainds, B, Binds)
5×5 Matrix{Float64}:
 0.0  0.0  0.0  0.0  0.0
 0.0  1.0  2.0  0.0  0.0
 0.0  3.0  4.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0
```

source

`Base.copy!` – Function.

```
copy!(dst, src) -> dst
```

In-place `copy` of `src` into `dst`, discarding any pre-existing elements in `dst`. If `dst` and `src` are of the same type, `dst == src` should hold after the call. If `dst` and `src` are vector types, they must have equal offset. If `dst` and `src` are multidimensional arrays, they must have equal `axes`.

Warning

Behavior can be unexpected when any mutated argument shares memory with any other argument.

See also `copyto!`.

Note

When operating on vector types, if `dst` and `src` are not of the same length, `dst` is resized to `length(src)` prior to the copy.

Julia 1.1

This method requires at least Julia 1.1. In Julia 1.0 this method is available from the `Future` standard library as `Future.copy!`.

source

```
copy!(dest::AbstractMatrix, src::UniformScaling)
```

Copies a `UniformScaling` onto a matrix.

Julia 1.12

This method is available as of Julia 1.12.

`Base.isassigned` – Function.

```
isassigned(array, i)::Bool
```

Test whether the given array has a value associated with index `i`. Return `false` if the index is out of bounds, or has an undefined reference.

Examples

```
julia> isassigned(rand(3, 3), 5)
true

julia> isassigned(rand(3, 3), 3 * 3 + 1)
false

julia> mutable struct Foo end

julia> v = similar(rand(3), Foo)
```

```
3-element Vector{Foo}:
 #undef
 #undef
 #undef

julia> isassigned(v, 1)
false
```

source

Base.isassigned – Method.

```
isassigned(v::Tuple, i::Integer) -> Bool
```

Return `true` if index `i` is within the bounds of tuple `v`, i.e. `1 ≤ i ≤ length(v)`.

Julia 1.13

This method requires at least Julia 1.13.

source

Base.Colon – Type.

```
Colon()
```

Colons (:) are used to signify indexing entire objects or dimensions at once.

Very few operations are defined on Colons directly; instead they are converted by `to_indices` to an internal vector type (`Base.Slice`) to represent the collection of indices they span before being used.

The singleton instance of `Colon` is also a function used to construct ranges; see `:`.

source

Base.IteratorsMD.CartesianIndex – Type.

```
CartesianIndex(i, j, k...)   -> I
CartesianIndex((i, j, k...)) -> I
```

Create a multidimensional index `I`, which can be used for indexing a multidimensional array `A`. In particular, `A[I]` is equivalent to `A[i,j,k...]`. One can freely mix integer and `CartesianIndex` indices; for example, `A[Ipre, i, Ipost]` (where `Ipre` and `Ipost` are `CartesianIndex` indices and `i` is an `Int`) can be a useful expression when writing algorithms that work along a single dimension of an array of arbitrary dimensionality.

A `CartesianIndex` is sometimes produced by `eachindex`, and always when iterating with an explicit `CartesianIndices`.

An `I::CartesianIndex` is treated as a "scalar" (not a container) for `broadcast`. In order to iterate over the components of a `CartesianIndex`, convert it to a tuple with `Tuple(I)`.

Examples

```
julia> A = reshape(Vector(1:16), (2, 2, 2, 2))
2×2×2×2 Array{Int64, 4}:
[:, :, 1, 1] =
 1  3
 2  4

[:, :, 2, 1] =
 5  7
 6  8

[:, :, 1, 2] =
  9  11
 10  12

[:, :, 2, 2] =
 13  15
 14  16

julia> A[CartesianIndex((1, 1, 1, 1))]
1

julia> A[CartesianIndex((1, 1, 1, 2))]
9

julia> A[CartesianIndex((1, 1, 2, 1))]
5
```

Julia 1.10

Using a `CartesianIndex` as a "scalar" for `broadcast` requires Julia 1.10; in previous releases, use `Ref(I)`.

source

`Base.IteratorsMD.CartesianIndices` – Type.

```
CartesianIndices(sz::Dims) -> R
CartesianIndices((istart:[istep:]istop, jstart:[jstep:]jstop, ...)) -> R
```

Define a region `R` spanning a multidimensional rectangular range of integer indices. These are most commonly encountered in the context of iteration, where `for I in R ... end` will return `CartesianIndex` indices `I` equivalent to the nested loops

```
for j = jstart:jstep:jstop
    for i = istart:istep:istop
        ...
    end
end
```

Consequently these can be useful for writing algorithms that work in arbitrary dimensions.

```
CartesianIndices(A::AbstractArray) -> R
```

As a convenience, constructing a `CartesianIndices` from an array makes a range of its indices.

Julia 1.6

The step range method `CartesianIndices((istart:istep:istop, jstart:[jstep:]jstop, ...))` requires at least Julia 1.6.

Examples

```
julia> foreach(println, CartesianIndices((2, 2, 2)))
CartesianIndex(1, 1, 1)
CartesianIndex(2, 1, 1)
CartesianIndex(1, 2, 1)
CartesianIndex(2, 2, 1)
CartesianIndex(1, 1, 2)
CartesianIndex(2, 1, 2)
CartesianIndex(1, 2, 2)
CartesianIndex(2, 2, 2)

julia> CartesianIndices(fill(1, (2,3)))
CartesianIndices((2, 3))
```

Conversion between linear and cartesian indices

Linear index to cartesian index conversion exploits the fact that a `CartesianIndices` is an `AbstractArray` and can be indexed linearly:

```
julia> cartesian = CartesianIndices((1:3, 1:2))
CartesianIndices((1:3, 1:2))

julia> cartesian[4]
CartesianIndex(1, 2)

julia> cartesian = CartesianIndices((1:2:5, 1:2))
CartesianIndices((1:2:5, 1:2))

julia> cartesian[2, 2]
CartesianIndex(3, 2)
```

Broadcasting

`CartesianIndices` support broadcasting arithmetic (+ and -) with a `CartesianIndex`.

Julia 1.1

Broadcasting of CartesianIndices requires at least Julia 1.1.

```
julia> CIs = CartesianIndices((2:3, 5:6))
CartesianIndices((2:3, 5:6))

julia> CI = CartesianIndex(3, 4)
CartesianIndex(3, 4)
```

```
julia> CIs .+ CI
CartesianIndices((5:6, 9:10))
```

For cartesian to linear index conversion, see LinearIndices.

source

Base.Dims – Type.

```
Dims{N}
```

An NTuple of N Ints used to represent the dimensions of an AbstractArray.

source

Base.LinearIndices – Type.

```
LinearIndices(A::AbstractArray)
```

Return a LinearIndices array with the same shape and axes as A, holding the linear index of each entry in A. Indexing this array with cartesian indices allows mapping them to linear indices.

For arrays with conventional indexing (indices start at 1), or any multidimensional array, linear indices range from 1 to length(A). However, for AbstractVectors linear indices are axes(A, 1), and therefore do not start at 1 for vectors with unconventional indexing.

Calling this function is the "safe" way to write algorithms that exploit linear indexing.

Examples

```
julia> A = fill(1, (5,6,7));

julia> b = LinearIndices(A);

julia> extrema(b)
(1, 210)
```

```
LinearIndices(inds::CartesianIndices) -> R
LinearIndices(sz::Dims) -> R
LinearIndices((istart:istop, jstart:jstop, ...)) -> R
```

Return a LinearIndices array with the specified shape or axes.

Examples

The main purpose of this constructor is intuitive conversion from cartesian to linear indexing:

```
julia> linear = LinearIndices((1:3, 1:2))
3×2 LinearIndices{2, Tuple{UnitRange{Int64}, UnitRange{Int64}}}:
 1  4
 2  5
 3  6

julia> linear[1,2]
4
```

source

Base.to_indices – Function.

```
to_indices(A, I::Tuple)
```

Convert the tuple `I` to a tuple of indices for use in indexing into array `A`.

The returned tuple must only contain either `Ints` or `AbstractArrays` of scalar indices that are supported by array `A`. It will error upon encountering a novel index type that it does not know how to process.

For simple index types, it defers to the unexported `Base.to_index(A, i)` to process each index `i`. While this internal function is not intended to be called directly, `Base.to_index` may be extended by custom array or index types to provide custom indexing behaviors.

More complicated index types may require more context about the dimension into which they index. To support those cases, `to_indices(A, I)` calls `to_indices(A, axes(A), I)`, which then recursively walks through both the given tuple of indices and the dimensional indices of `A` in tandem. As such, not all index types are guaranteed to propagate to `Base.to_index`.

Examples

```
julia> A = zeros(1,2,3,4);

julia> to_indices(A, (1,1,2,2))
(1, 1, 2, 2)

julia> to_indices(A, (1,1,2,20)) # no bounds checking
(1, 1, 2, 20)

julia> to_indices(A, (CartesianIndex((1,)), 2, CartesianIndex((3,4)))) # exotic index
(1, 2, 3, 4)

julia> to_indices(A, ([1,1], 1:2, 3, 4))
([1, 1], 1:2, 3, 4)

julia> to_indices(A, (1,2)) # no shape checking
(1, 2)
```

source

Base.checkbounds – Function.

```
checkbounds(A, I...)
```

Throw an error if the specified indices `I` are not in bounds for the given array `A`.

source

```
checkbounds(Bool, A, I...)
```

Return `true` if the specified indices `I` are in bounds for the given array `A`. Subtypes of `AbstractArray` should specialize this method if they need to provide custom bounds checking behaviors; however, in many cases one can rely on `A`'s indices and `checkindex`.

See also checkindex.

Examples

```
julia> A = rand(3, 3);

julia> checkbounds(Bool, A, 2)
true

julia> checkbounds(Bool, A, 3, 4)
false

julia> checkbounds(Bool, A, 1:3)
true

julia> checkbounds(Bool, A, 1:3, 2:4)
false
```

source

Base.checkindex – Function.

```
checkindex(Bool, inds::AbstractUnitRange, index)
```

Return true if the given index is within the bounds of inds. Custom types that would like to behave as indices for all arrays can extend this method in order to provide a specialized bounds checking implementation.

See also checkbounds.

Examples

```
julia> checkindex(Bool, 1:20, 8)
true

julia> checkindex(Bool, 1:20, 21)
false
```

source

Base.elsize – Function.

```
elsize(type)
```

Compute the memory stride in bytes between consecutive elements of eltype stored inside the given type, if the array elements are stored densely with a uniform linear stride.

Examples

```
julia> Base.elsize(rand(Float32, 10))
4
```

source

While most code can be written in an index-agnostic manner (see, e.g., eachindex), it can sometimes be useful to explicitly check for offset axes:

Base.require_one_based_indexing – Function.

```
require_one_based_indexing(A::AbstractArray)
require_one_based_indexing(A,B...)
```

Throw an ArgumentError if the indices of any argument start with something other than 1 along any axis. See also has_offset_axes.

Julia 1.2

This function requires at least Julia 1.2.

source

Base.has_offset_axes – Function.

```
has_offset_axes(A)
has_offset_axes(A, B, ...)
```

Return true if the indices of A start with something other than 1 along any axis. If multiple arguments are passed, equivalent to has_offset_axes(A) || has_offset_axes(B) || ....

See also require_one_based_indexing.

source

49.5 Views (SubArrays and other view types)

A "view" is a data structure that acts like an array (it is a subtype of AbstractArray), but the underlying data is actually part of another array.

For example, if x is an array and v = @view x[1:10], then v acts like a 10-element array, but its data is actually accessing the first 10 elements of x. Writing to a view, e.g. v[3] = 2, writes directly to the underlying array x (in this case modifying x[3]).

Slicing operations like x[1:10] create a copy by default in Julia. @view x[1:10] changes it to make a view. The @views macro can be used on a whole block of code (e.g. @views function foo() .... end or @views begin ... end) to change all the slicing operations in that block to use views. Sometimes making a copy of the data is faster and sometimes using a view is faster, as described in the performance tips.

Base.view – Function.

```
view(A, inds...)
```

Like getindex, but returns a lightweight array that lazily references (or is effectively a *view* into) the parent array A at the given index or indices inds instead of eagerly extracting elements or constructing a copied subset. Calling getindex or setindex! on the returned value (often a SubArray) computes the indices to access or modify the parent array on the fly. The behavior is undefined if the shape of the parent array is changed after view is called because there is no bound check for the parent array; e.g., it may cause a segmentation fault. It is likewise undefined behavior to modify the inds array(s) after construction of the view.

Some immutable parent arrays (like ranges) may choose to simply recompute a new array in some circumstances instead of returning a `SubArray` if doing so is efficient and provides compatible semantics.

Julia 1.6

In Julia 1.6 or later, `view` can be called on an `AbstractString`, returning a `SubString`.

Examples

```
julia> A = [1 2; 3 4]
2×2 Matrix{Int64}:
 1  2
 3  4

julia> b = view(A, :, 1)
2-element view(::Matrix{Int64}, :, 1) with eltype Int64:
 1
 3

julia> fill!(b, 0)
2-element view(::Matrix{Int64}, :, 1) with eltype Int64:
 0
 0

julia> A # Note A has changed even though we modified b
2×2 Matrix{Int64}:
 0  2
 0  4

julia> view(2:5, 2:3) # returns a range as type is immutable
3:4
```

source

Base.@view – Macro.

```
@view A[inds...]
```

Transform the indexing expression `A[inds...]` into the equivalent `view` call.

This can only be applied directly to a single indexing expression and is particularly helpful for expressions that include the special `begin` or `end` indexing syntaxes like `A[begin, 2:end-1]` (as those are not supported by the normal `view` function).

Note that `@view` cannot be used as the target of a regular assignment (e.g., `@view(A[1, 2:end]) = ...`), nor would the un-decorated indexed assignment (`A[1, 2:end] = ...`) or broadcasted indexed assignment (`A[1, 2:end] .= ...`) make a copy. It can be useful, however, for *updating* broadcasted assignments like `@view(A[1, 2:end]) .+= 1` because this is a simple syntax for `@view(A[1, 2:end]) .= @view(A[1, 2:end]) + 1`, and the indexing expression on the right-hand side would otherwise make a copy without the `@view`.

See also `@views` to switch an entire block of code to use views for non-scalar indexing.

Julia 1.5

Using `begin` in an indexing expression to refer to the first index requires at least Julia 1.5.

Examples

```
julia> A = [1 2; 3 4]
2×2 Matrix{Int64}:
 1  2
 3  4

julia> b = @view A[:, 1]
2-element view(::Matrix{Int64}, :, 1) with eltype Int64:
 1
 3

julia> fill!(b, 0)
2-element view(::Matrix{Int64}, :, 1) with eltype Int64:
 0
 0

julia> A
2×2 Matrix{Int64}:
 0  2
 0  4
```

source

`Base.@views` – Macro.

```
@views expression
```

Convert every array-slicing operation in the given expression (which may be a `begin`/`end` block, loop, function, etc.) to return a view. Scalar indices, non-array types, and explicit `getindex` calls (as opposed to `array[...]`) are unaffected.

Similarly, `@views` converts string slices into `SubString` views.

Note

The `@views` macro only affects `array[...]` expressions that appear explicitly in the given `expression`, not array slicing that occurs in functions called by that code.

Julia 1.5

Using `begin` in an indexing expression to refer to the first index was implemented in Julia 1.4, but was only supported by `@views` starting in Julia 1.5.

Examples

```
julia> A = zeros(3, 3);

julia> @views for row in 1:3
```

```
            b = A[row, :] # b is a view, not a copy
            b .= row      # assign every element to the row index
        end

julia> A
3×3 Matrix{Float64}:
 1.0  1.0  1.0
 2.0  2.0  2.0
 3.0  3.0  3.0
```

source

Base.parent – Function.

```
parent(A)
```

Return the underlying parent object of the view. This parent of objects of types SubArray, SubString, ReshapedArray or LinearAlgebra.Transpose is what was passed as an argument to view, reshape, transpose, etc. during object creation. If the input is not a wrapped object, return the input itself. If the input is wrapped multiple times, only the outermost wrapper will be removed.

Examples

```
julia> A = [1 2; 3 4]
2×2 Matrix{Int64}:
 1  2
 3  4

julia> V = view(A, 1:2, :)
2×2 view(::Matrix{Int64}, 1:2, :) with eltype Int64:
 1  2
 3  4

julia> parent(V)
2×2 Matrix{Int64}:
 1  2
 3  4
```

source

Base.parentindices – Function.

```
parentindices(A)
```

Return the indices in the parent which correspond to the view A.

Examples

```
julia> A = [1 2; 3 4];

julia> V = view(A, 1, :)
2-element view(::Matrix{Int64}, 1, :) with eltype Int64:
 1
 2
```

```
julia> parentindices(V)
(1, Base.Slice(Base.OneTo(2)))
```

source

Base.selectdim – Function.

```
selectdim(A, d::Integer, i)
```

Return a view of all the data of A where the index for dimension d equals i.

Equivalent to view(A,:,:,...,i,:,:,...) where i is in position d.

See also: eachslice.

Examples

```
julia> A = [1 2 3 4; 5 6 7 8]
2×4 Matrix{Int64}:
 1  2  3  4
 5  6  7  8

julia> selectdim(A, 2, 3)
2-element view(::Matrix{Int64}, :, 3) with eltype Int64:
 3
 7

julia> selectdim(A, 2, 3:4)
2×2 view(::Matrix{Int64}, :, 3:4) with eltype Int64:
 3  4
 7  8
```

source

Base.reinterpret – Function.

```
reinterpret(reshape, T, A::AbstractArray{S}) -> B
```

Change the type-interpretation of A while consuming or adding a "channel dimension."

If sizeof(T) = n*sizeof(S) for n>1, A's first dimension must be of size n and B lacks A's first dimension. Conversely, if sizeof(S) = n*sizeof(T) for n>1, B gets a new first dimension of size n. The dimensionality is unchanged if sizeof(T) == sizeof(S).

Julia 1.6

This method requires at least Julia 1.6.

Examples

```
julia> A = [1 2; 3 4]
2×2 Matrix{Int64}:
 1  2
 3  4
```

```
julia> reinterpret(reshape, Complex{Int}, A)    # the result is a vector
2-element reinterpret(reshape, Complex{Int64}, ::Matrix{Int64}) with eltype Complex{Int64}:
 1 + 3im
 2 + 4im

julia> a = [(1,2,3), (4,5,6)]
2-element Vector{Tuple{Int64, Int64, Int64}}:
 (1, 2, 3)
 (4, 5, 6)

julia> reinterpret(reshape, Int, a)             # the result is a matrix
3×2 reinterpret(reshape, Int64, ::Vector{Tuple{Int64, Int64, Int64}}) with eltype Int64:
 1  4
 2  5
 3  6
```

source

```
reinterpret(T::DataType, A::AbstractArray)
```

Construct a view of the array with the same binary data as the given array, but with T as element type.

This function also works on "lazy" array whose elements are not computed until they are explicitly retrieved. For instance, `reinterpret` on the range `1:6` works similarly as on the dense vector `collect(1:6)`:

```
julia> reinterpret(Float32, UInt32[1 2 3 4 5])
1×5 reinterpret(Float32, ::Matrix{UInt32}):
 1.0f-45  3.0f-45  4.0f-45  6.0f-45  7.0f-45

julia> reinterpret(Complex{Int}, 1:6)
3-element reinterpret(Complex{Int64}, ::UnitRange{Int64}):
 1 + 2im
 3 + 4im
 5 + 6im
```

If the location of padding bits does not line up between T and `eltype(A)`, the resulting array will be read-only or write-only, to prevent invalid bits from being written to or read from, respectively.

```
julia> a = reinterpret(Tuple{UInt8, UInt32}, UInt32[1, 2])
1-element reinterpret(Tuple{UInt8, UInt32}, ::Vector{UInt32}):
 (0x01, 0x00000002)

julia> a[1] = 3
ERROR: Padding of type Tuple{UInt8, UInt32} is not compatible with type UInt32.

julia> b = reinterpret(UInt32, Tuple{UInt8, UInt32}[(0x01, 0x00000002)]); # showing will error

julia> b[1]
ERROR: Padding of type UInt32 is not compatible with type Tuple{UInt8, UInt32}.
```

source

```
reinterpret(::Type{Out}, x::In)
```

Change the type-interpretation of the binary data in the isbits value `x` to that of the isbits type `Out`. The size (ignoring padding) of `Out` has to be the same as that of the type of `x`. For example, `reinterpret(Float32, UInt32(7))` interprets the 4 bytes corresponding to `UInt32(7)` as a `Float32`. Note that `reinterpret(In, reinterpret(Out, x)) === x`

```
julia> reinterpret(Float32, UInt32(7))
1.0f-44

julia> reinterpret(NTuple{2, UInt8}, 0x1234)
(0x34, 0x12)

julia> reinterpret(UInt16, (0x34, 0x12))
0x1234

julia> reinterpret(Tuple{UInt16, UInt8}, (0x01, 0x0203))
(0x0301, 0x02)
```

Note

The treatment of padding differs from reinterpret(::DataType, ::AbstractArray).

Warning

Use caution if some combinations of bits in `Out` are not considered valid and would otherwise be prevented by the type's constructors and methods. Unexpected behavior may result without additional validation.

source

`Base.reshape` – Function.

```
reshape(A, dims...)::AbstractArray
reshape(A, dims)::AbstractArray
```

Return an array with the same data as `A`, but with different dimension sizes or number of dimensions. The two arrays share the same underlying data, so that the result is mutable if and only if `A` is mutable, and setting elements of one alters the values of the other.

The new dimensions may be specified either as a list of arguments or as a shape tuple. At most one dimension may be specified with a `:`, in which case its length is computed such that its product with all the specified dimensions is equal to the length of the original array `A`. The total number of elements must not change.

Examples

```
julia> A = Vector(1:16)
16-element Vector{Int64}:
  1
  2
  3
  4
```

```
 5
 6
 7
 8
 9
10
11
12
13
14
15
16

julia> reshape(A, (4, 4))
4×4 Matrix{Int64}:
 1  5   9  13
 2  6  10  14
 3  7  11  15
 4  8  12  16

julia> reshape(A, 2, :)
2×8 Matrix{Int64}:
 1  3  5  7   9  11  13  15
 2  4  6  8  10  12  14  16

julia> reshape(1:6, 2, 3)
2×3 reshape(::UnitRange{Int64}, 2, 3) with eltype Int64:
 1  3  5
 2  4  6
```

source

`Base.insertdims` – Function.

```
insertdims(A; dims)
```

Inverse of `dropdims`; return an array with new singleton dimensions at every dimension in `dims`.

Repeated dimensions are forbidden and the largest entry in `dims` must be less than or equal than `ndims(A) + length(dims)`.

The result shares the same underlying data as `A`, such that the result is mutable if and only if `A` is mutable, and setting elements of one alters the values of the other.

See also: `dropdims`, `reshape`, `vec`.

Examples

```
julia> x = [1 2 3; 4 5 6]
2×3 Matrix{Int64}:
 1  2  3
 4  5  6

julia> insertdims(x, dims=3)
2×3×1 Array{Int64, 3}:
```

```
[:, :, 1] =
 1  2  3
 4  5  6

julia> insertdims(x, dims=(1,2,5)) == reshape(x, 1, 1, 2, 3, 1)
true

julia> dropdims(insertdims(x, dims=(1,2,5)), dims=(1,2,5))
2×3 Matrix{Int64}:
 1  2  3
 4  5  6
```

Julia 1.12

Requires Julia 1.12 or later.

source

Base.dropdims – Function.

```
dropdims(A; dims)
```

Return an array with the same data as `A`, but with the dimensions specified by `dims` removed. `size(A,d)` must equal 1 for every `d` in `dims`, and repeated dimensions or numbers outside `1:ndims(A)` are forbidden.

The result shares the same underlying data as `A`, such that the result is mutable if and only if `A` is mutable, and setting elements of one alters the values of the other.

See also: reshape, vec.

Examples

```
julia> a = reshape(Vector(1:4),(2,2,1,1))
2×2×1×1 Array{Int64, 4}:
[:, :, 1, 1] =
 1  3
 2  4

julia> b = dropdims(a; dims=3)
2×2×1 Array{Int64, 3}:
[:, :, 1] =
 1  3
 2  4

julia> b[1,1,1] = 5; a
2×2×1×1 Array{Int64, 4}:
[:, :, 1, 1] =
 5  3
 2  4
```

source

Base.vec – Function.

```
vec(a::AbstractArray)::AbstractVector
```

Reshape the array a as a one-dimensional column vector. Return a if it is already an `AbstractVector`. The resulting array shares the same underlying data as a, so it will only be mutable if a is mutable, in which case modifying one will also modify the other.

Examples

```
julia> a = [1 2 3; 4 5 6]
2×3 Matrix{Int64}:
 1  2  3
 4  5  6

julia> vec(a)
6-element Vector{Int64}:
 1
 4
 2
 5
 3
 6

julia> vec(1:3)
1:3
```

See also `reshape`, `dropdims`.

source

`Base.SubArray` – Type.

```
SubArray{T,N,P,I,L} <: AbstractArray{T,N}
```

N-dimensional view into a parent array (of type `P`) with an element type `T`, restricted by a tuple of indices (of type `I`). `L` is true for types that support fast linear indexing, and `false` otherwise.

Construct SubArrays using the `view` function.

source

49.6 Concatenation and permutation

`Base.cat` – Function.

```
cat(A...; dims)
```

Concatenate the input arrays along the dimensions specified in `dims`.

Along a dimension `d in dims`, the size of the output array is `sum(size(a,d) for a in A)`. Along other dimensions, all input arrays should have the same size, which will also be the size of the output array along those dimensions.

If `dims` is a single number, the different arrays are tightly packed along that dimension. If `dims` is an iterable containing several dimensions, the positions along these dimensions are increased simultaneously for each input array, filling with zero elsewhere. This allows one to construct block-diagonal matrices as `cat(matrices...; dims=(1,2))`, and their higher-dimensional analogues.

The special case dims=1 is vcat, and dims=2 is hcat. See also hvcat, hvncat, stack, repeat.

The keyword also accepts Val(dims).

Julia 1.8

For multiple dimensions dims = Val(::Tuple) was added in Julia 1.8.

Examples

Concatenate two arrays in different dimensions:

```
julia> a = [1 2 3]
1×3 Matrix{Int64}:
 1  2  3

julia> b = [4 5 6]
1×3 Matrix{Int64}:
 4  5  6

julia> cat(a, b; dims=1)
2×3 Matrix{Int64}:
 1  2  3
 4  5  6

julia> cat(a, b; dims=2)
1×6 Matrix{Int64}:
 1  2  3  4  5  6

julia> cat(a, b; dims=(1, 2))
2×6 Matrix{Int64}:
 1  2  3  0  0  0
 0  0  0  4  5  6
```

Extended Help

Concatenate 3D arrays:

```
julia> a = ones(2, 2, 3);

julia> b = ones(2, 2, 4);

julia> c = cat(a, b; dims=3);

julia> size(c) == (2, 2, 7)
true
```

Concatenate arrays of different sizes:

```
julia> cat([1 2; 3 4], [pi, pi], fill(10, 2,3,1); dims=2)  # same as hcat
2×6×1 Array{Float64, 3}:
[:, :, 1] =
 1.0  2.0  3.14159  10.0  10.0  10.0
 3.0  4.0  3.14159  10.0  10.0  10.0
```

Construct a block diagonal matrix:

```
julia> cat(true, trues(2,2), trues(4)', dims=(1,2))  # block-diagonal
4×7 Matrix{Bool}:
 1  0  0  0  0  0  0
 0  1  1  0  0  0  0
 0  1  1  0  0  0  0
 0  0  0  1  1  1  1
```

```
julia> cat(1, [2], [3;;]; dims=Val(2))
1×3 Matrix{Int64}:
 1  2  3
```

Note

cat does not join two strings, you may want to use *.

```
julia> a = "aaa";

julia> b = "bbb";

julia> cat(a, b; dims=1)
2-element Vector{String}:
 "aaa"
 "bbb"

julia> cat(a, b; dims=2)
1×2 Matrix{String}:
 "aaa"  "bbb"

julia> a * b
"aaabbb"
```

source

Base.vcat – Function.

```
vcat(A...)
```

Concatenate arrays or numbers vertically. Equivalent to cat(A...; dims=1), and to the syntax [a; b; c].

To concatenate a large vector of arrays, reduce(vcat, A) calls an efficient method when A isa AbstractVector{<:Abstract rather than working pairwise.

See also hcat, Iterators.flatten, stack.

Examples

```
julia> v = vcat([1,2], [3,4])
4-element Vector{Int64}:
 1
 2
```

```
 3
 4

julia> v == vcat(1, 2, [3,4])  # accepts numbers
true

julia> v == [1; 2; [3,4]]  # syntax for the same operation
true

julia> summary(ComplexF64[1; 2; [3,4]])  # syntax for supplying the element type
"4-element Vector{ComplexF64}"

julia> vcat(range(1, 2, length=3))  # collects lazy ranges
3-element Vector{Float64}:
 1.0
 1.5
 2.0

julia> two = ([10, 20, 30]', Float64[4 5 6; 7 8 9])  # row vector and a matrix
(adjoint([10, 20, 30]), [4.0 5.0 6.0; 7.0 8.0 9.0])

julia> vcat(two...)
3×3 Matrix{Float64}:
 10.0  20.0  30.0
  4.0   5.0   6.0
  7.0   8.0   9.0

julia> vs = [[1, 2], [3, 4], [5, 6]];

julia> reduce(vcat, vs)  # more efficient than vcat(vs...)
6-element Vector{Int64}:
 1
 2
 3
 4
 5
 6

julia> ans == collect(Iterators.flatten(vs))
true
```

source

Base.hcat – Function.

```
hcat(A...)
```

Concatenate arrays or numbers horizontally. Equivalent to `cat(A...; dims=2)`, and to the syntax `[a b c]` or `[a;; b;; c]`.

For a large vector of arrays, `reduce(hcat, A)` calls an efficient method when `A isa AbstractVector{<:AbstractVecOrMat}`. For a vector of vectors, this can also be written `stack(A)`.

See also `vcat`, `hvcat`.

Examples

```
julia> hcat([1,2], [3,4], [5,6])
2×3 Matrix{Int64}:
 1  3  5
 2  4  6

julia> hcat(1, 2, [30 40], [5, 6, 7]')  # accepts numbers
1×7 Matrix{Int64}:
 1  2  30  40  5  6  7

julia> ans == [1 2 [30 40] [5, 6, 7]']  # syntax for the same operation
true

julia> Float32[1 2 [30 40] [5, 6, 7]']  # syntax for supplying the eltype
1×7 Matrix{Float32}:
 1.0  2.0  30.0  40.0  5.0  6.0  7.0

julia> ms = [zeros(2,2), [1 2; 3 4], [50 60; 70 80]];

julia> reduce(hcat, ms)  # more efficient than hcat(ms...)
2×6 Matrix{Float64}:
 0.0  0.0  1.0  2.0  50.0  60.0
 0.0  0.0  3.0  4.0  70.0  80.0

julia> stack(ms) |> summary  # disagrees on a vector of matrices
"2×2×3 Array{Float64, 3}"

julia> hcat(Int[], Int[], Int[])  # empty vectors, each of size (0,)
0×3 Matrix{Int64}

julia> hcat([1.1, 9.9], Matrix(undef, 2, 0))  # hcat with empty 2×0 Matrix
2×1 Matrix{Any}:
 1.1
 9.9
```

source

Base.hvcat – Function.

```
hvcat(blocks_per_row::Union{Tuple{Vararg{Int}}, Int}, values...)
```

Horizontal and vertical concatenation in one call. This function is called for block matrix syntax. The first argument specifies the number of arguments to concatenate in each block row. If the first argument is a single integer n, then all block rows are assumed to have n block columns.

Examples

```
julia> a, b, c, d, e, f = 1, 2, 3, 4, 5, 6
(1, 2, 3, 4, 5, 6)

julia> [a b c; d e f]
2×3 Matrix{Int64}:
 1  2  3
```

```
 4  5  6

julia> hvcat((3,3), a,b,c,d,e,f)
2×3 Matrix{Int64}:
 1  2  3
 4  5  6

julia> [a b; c d; e f]
3×2 Matrix{Int64}:
 1  2
 3  4
 5  6

julia> hvcat((2,2,2), a,b,c,d,e,f)
3×2 Matrix{Int64}:
 1  2
 3  4
 5  6
julia> hvcat((2,2,2), a,b,c,d,e,f) == hvcat(2, a,b,c,d,e,f)
true
```

source

Base.hvncat – Function.

```
hvncat(dim::Int, row_first, values...)
hvncat(dims::Tuple{Vararg{Int}}, row_first, values...)
hvncat(shape::Tuple{Vararg{Tuple}}, row_first, values...)
```

Horizontal, vertical, and n-dimensional concatenation of many `values` in one call.

This function is called for block matrix syntax. The first argument either specifies the shape of the concatenation, similar to `hvcat`, as a tuple of tuples, or the dimensions that specify the key number of elements along each axis, and is used to determine the output dimensions. The `dims` form is more performant, and is used by default when the concatenation operation has the same number of elements along each axis (e.g., [a b; c d;;; e f ; g h]). The `shape` form is used when the number of elements along each axis is unbalanced (e.g., [a b ; c]). Unbalanced syntax needs additional validation overhead. The `dim` form is an optimization for concatenation along just one dimension. `row_first` indicates how `values` are ordered. The meaning of the first and second elements of `shape` are also swapped based on `row_first`.

Examples

```
julia> a, b, c, d, e, f = 1, 2, 3, 4, 5, 6
(1, 2, 3, 4, 5, 6)

julia> [a b c;;; d e f]
1×3×2 Array{Int64, 3}:
[:, :, 1] =
 1  2  3

[:, :, 2] =
 4  5  6
```

```
julia> hvncat((2,1,3), false, a,b,c,d,e,f)
2×1×3 Array{Int64, 3}:
[:, :, 1] =
 1
 2

[:, :, 2] =
 3
 4

[:, :, 3] =
 5
 6

julia> [a b;;; c d;;; e f]
1×2×3 Array{Int64, 3}:
[:, :, 1] =
 1  2

[:, :, 2] =
 3  4

[:, :, 3] =
 5  6

julia> hvncat(((3, 3), (3, 3), (6,)), true, a, b, c, d, e, f)
1×3×2 Array{Int64, 3}:
[:, :, 1] =
 1  2  3

[:, :, 2] =
 4  5  6
```

Examples for construction of the arguments

```
[a b c ; d e f ;;;
 g h i ; j k l ;;;
 m n o ; p q r ;;;
 s t u ; v w x]
⇒ dims = (2, 3, 4)

[a b ; c ;;; d ;;;;]
 ___   _     _
 2     1     1 = elements in each row (2, 1, 1)
 _______     _
 3           1 = elements in each column (3, 1)
 _____________
 4             = elements in each 3d slice (4,)
 _____________
 4             = elements in each 4d slice (4,)
⇒ shape = ((2, 1, 1), (3, 1), (4,), (4,)) with `row_first` = true
```

source

Base.stack – Function.

```
stack(f, args...; [dims])
```

Apply a function to each element of a collection, and `stack` the result. Or to several collections, `zip`ped together.

The function should return arrays (or tuples, or other iterators) all of the same size. These become slices of the result, each separated along `dims` (if given) or by default along the last dimensions.

See also `mapslices`, `eachcol`.

Examples

```
julia> stack(c -> (c, c-32), "julia")
2×5 Matrix{Char}:
 'j'  'u'  'l'  'i'  'a'
 'J'  'U'  'L'  'I'  'A'

julia> stack(eachrow([1 2 3; 4 5 6]), (10, 100); dims=1) do row, n
         vcat(row, row .* n, row ./ n)
       end
2×9 Matrix{Float64}:
 1.0  2.0  3.0   10.0   20.0   30.0  0.1   0.2   0.3
 4.0  5.0  6.0  400.0  500.0  600.0  0.04  0.05  0.06
```

source

```
stack(iter; [dims])
```

Combine a collection of arrays (or other iterable objects) of equal size into one larger array, by arranging them along one or more new dimensions.

By default the axes of the elements are placed first, giving `size(result) = (size(first(iter))..., size(iter)...)`. This has the same order of elements as `Iterators.flatten(iter)`.

With keyword `dims::Integer`, instead the `i`th element of `iter` becomes the slice `selectdim(result, dims, i)`, so that `size(result, dims) == length(iter)`. In this case `stack` reverses the action of `eachslice` with the same `dims`.

The various `cat` functions also combine arrays. However, these all extend the arrays' existing (possibly trivial) dimensions, rather than placing the arrays along new dimensions. They also accept arrays as separate arguments, rather than a single collection.

Julia 1.9

This function requires at least Julia 1.9.

Examples

```
julia> vecs = (1:2, [30, 40], Float32[500, 600]);

julia> mat = stack(vecs)
2×3 Matrix{Float32}:
 1.0  30.0  500.0
```

```
 2.0  40.0  600.0

julia> mat == hcat(vecs...) == reduce(hcat, collect(vecs))
true

julia> vec(mat) == vcat(vecs...) == reduce(vcat, collect(vecs))
true

julia> stack(zip(1:4, 10:99))  # accepts any iterators of iterators
2×4 Matrix{Int64}:
  1   2   3   4
 10  11  12  13

julia> vec(ans) == collect(Iterators.flatten(zip(1:4, 10:99)))
true

julia> stack(vecs; dims=1)  # unlike any cat function, 1st axis of vecs[1] is 2nd axis of
↪  result
3×2 Matrix{Float32}:
   1.0    2.0
  30.0   40.0
 500.0  600.0

julia> x = rand(3,4);

julia> x == stack(eachcol(x)) == stack(eachrow(x), dims=1)  # inverse of eachslice
true
```

Higher-dimensional examples:

```
julia> A = rand(5, 7, 11);

julia> E = eachslice(A, dims=2);  # a vector of matrices

julia> (element = size(first(E)), container = size(E))
(element = (5, 11), container = (7,))

julia> stack(E) |> size
(5, 11, 7)

julia> stack(E) == stack(E; dims=3) == cat(E...; dims=3)
true

julia> A == stack(E; dims=2)
true

julia> M = (fill(10i+j, 2, 3) for i in 1:5, j in 1:7);

julia> (element = size(first(M)), container = size(M))
(element = (2, 3), container = (5, 7))

julia> stack(M) |> size  # keeps all dimensions
(2, 3, 5, 7)
```

```
julia> stack(M; dims=1) |> size  # vec(container) along dims=1
(35, 2, 3)

julia> hvcat(5, M...) |> size  # hvcat puts matrices next to each other
(14, 15)
```

source

Base.vect – Function.

```
vect(X...)
```

Create a Vector with element type computed from the promote_typeof of the argument, containing the argument list.

Examples

```
julia> a = Base.vect(UInt8(1), 2.5, 1//2)
3-element Vector{Float64}:
 1.0
 2.5
 0.5
```

source

Base.circshift – Function.

```
circshift(A, shifts)
```

Circularly shift, i.e. rotate, the data in A. The second argument is a tuple or vector giving the amount to shift in each dimension, or an integer to shift only in the first dimension.

The generated code is most efficient when the shift amounts are known at compile-time, i.e., compile-time constants.

See also: circshift!, circcopy!, bitrotate, <<.

Examples

```
julia> b = reshape(Vector(1:16), (4,4))
4×4 Matrix{Int64}:
 1  5   9  13
 2  6  10  14
 3  7  11  15
 4  8  12  16

julia> circshift(b, (0,2))
4×4 Matrix{Int64}:
  9  13  1  5
 10  14  2  6
 11  15  3  7
 12  16  4  8

julia> circshift(b, (-1,0))
```

```
4×4 Matrix{Int64}:
 2  6  10  14
 3  7  11  15
 4  8  12  16
 1  5   9  13

julia> a = BitArray([true, true, false, false, true])
5-element BitVector:
 1
 1
 0
 0
 1

julia> circshift(a, 1)
5-element BitVector:
 1
 1
 1
 0
 0

julia> circshift(a, -1)
5-element BitVector:
 1
 0
 0
 1
 1

julia> x = (1, 2, 3, 4, 5)
(1, 2, 3, 4, 5)

julia> circshift(x, 4)
(2, 3, 4, 5, 1)

julia> z = (1, 'a', -7.0, 3)
(1, 'a', -7.0, 3)

julia> circshift(z, -1)
('a', -7.0, 3, 1)
```

source

Base.circshift! – Function.

```
circshift!(dest, src, shifts)
```

Circularly shift, i.e. rotate, the data in `src`, storing the result in `dest`. `shifts` specifies the amount to shift in each dimension.

Warning

Behavior can be unexpected when any mutated argument shares memory with any other argument.

See also circshift.

source

```
circshift!(a::AbstractVector, shift::Integer)
```

Circularly shift, or rotate, the data in vector a by shift positions.

Examples

```
julia> circshift!([1, 2, 3, 4, 5], 2)
5-element Vector{Int64}:
 4
 5
 1
 2
 3

julia> circshift!([1, 2, 3, 4, 5], -2)
5-element Vector{Int64}:
 3
 4
 5
 1
 2
```

source

Base.circcopy! – Function.

```
circcopy!(dest, src)
```

Copy src to dest, indexing each dimension modulo its length. src and dest must have the same size, but can be offset in their indices; any offset results in a (circular) wraparound. If the arrays have overlapping indices, then on the domain of the overlap dest agrees with src.

Warning

Behavior can be unexpected when any mutated argument shares memory with any other argument.

See also: circshift.

Examples

```
julia> src = reshape(Vector(1:16), (4,4))
4×4 Matrix{Int64}:
 1  5   9  13
 2  6  10  14
```

```
 3  7  11  15
 4  8  12  16

julia> dest = OffsetArray{Int}(undef, (0:3,2:5));

julia> circcopy!(dest, src)
4×4 OffsetArray(::Matrix{Int64}, 0:3, 2:5) with eltype Int64 with indices 0:3×2:5:
 8  12  16  4
 5   9  13  1
 6  10  14  2
 7  11  15  3

julia> dest[1:3,2:4] == src[1:3,2:4]
true
```

source

Base.findall – Method.

```
findall(A)
```

Return a vector `I` of the `true` indices or keys of `A`. If there are no such elements of `A`, return an empty array. To search for other kinds of values, pass a predicate as the first argument.

Indices or keys are of the same type as those returned by `keys(A)` and `pairs(A)`.

See also: `findfirst`, `searchsorted`.

Examples

```
julia> A = [true, false, false, true]
4-element Vector{Bool}:
 1
 0
 0
 1

julia> findall(A)
2-element Vector{Int64}:
 1
 4

julia> A = [true false; false true]
2×2 Matrix{Bool}:
 1  0
 0  1

julia> findall(A)
2-element Vector{CartesianIndex{2}}:
 CartesianIndex(1, 1)
 CartesianIndex(2, 2)

julia> findall(falses(3))
Int64[]
```

source

Base.findall – Method.

```
findall(f::Function, A)
```

Return a vector I of the indices or keys of A where f(A[I]) returns true. If there are no such elements of A, return an empty array.

Indices or keys are of the same type as those returned by keys(A) and pairs(A).

Examples

```
julia> x = [1, 3, 4]
3-element Vector{Int64}:
 1
 3
 4

julia> findall(isodd, x)
2-element Vector{Int64}:
 1
 2

julia> A = [1 2 0; 3 4 0]
2×3 Matrix{Int64}:
 1  2  0
 3  4  0
julia> findall(isodd, A)
2-element Vector{CartesianIndex{2}}:
 CartesianIndex(1, 1)
 CartesianIndex(2, 1)

julia> findall(!iszero, A)
4-element Vector{CartesianIndex{2}}:
 CartesianIndex(1, 1)
 CartesianIndex(2, 1)
 CartesianIndex(1, 2)
 CartesianIndex(2, 2)

julia> d = Dict(:A => 10, :B => -1, :C => 0)
Dict{Symbol, Int64} with 3 entries:
  :A => 10
  :B => -1
  :C => 0

julia> findall(≥(0), d)
2-element Vector{Symbol}:
 :A
 :C
```

source

Base.findfirst – Method.

```
findfirst(A)
```

Return the index or key of the first `true` value in `A`. Return `nothing` if no such value is found. To search for other kinds of values, pass a predicate as the first argument.

Indices or keys are of the same type as those returned by `keys(A)` and `pairs(A)`.

See also: `findall`, `findnext`, `findlast`, `searchsortedfirst`.

Examples

```
julia> A = [false, false, true, false]
4-element Vector{Bool}:
 0
 0
 1
 0

julia> findfirst(A)
3

julia> findfirst(falses(3)) # returns nothing, but not printed in the REPL

julia> A = [false false; true false]
2×2 Matrix{Bool}:
 0  0
 1  0

julia> findfirst(A)
CartesianIndex(2, 1)
```

source

`Base.findfirst` – Method.

```
findfirst(predicate::Function, A)
```

Return the index or key of the first element of `A` for which `predicate` returns `true`. Return `nothing` if there is no such element.

Indices or keys are of the same type as those returned by `keys(A)` and `pairs(A)`.

Examples

```
julia> A = [1, 4, 2, 2]
4-element Vector{Int64}:
 1
 4
 2
 2

julia> findfirst(iseven, A)
2

julia> findfirst(x -> x>10, A) # returns nothing, but not printed in the REPL
```

```
julia> findfirst(isequal(4), A)
2

julia> A = [1 4; 2 2]
2×2 Matrix{Int64}:
 1  4
 2  2

julia> findfirst(iseven, A)
CartesianIndex(2, 1)
```

source

Base.findlast – Method.

```
findlast(A)
```

Return the index or key of the last `true` value in `A`. Return `nothing` if there is no `true` value in `A`.

Indices or keys are of the same type as those returned by `keys(A)` and `pairs(A)`.

See also: `findfirst`, `findprev`, `findall`.

Examples

```
julia> A = [true, false, true, false]
4-element Vector{Bool}:
 1
 0
 1
 0

julia> findlast(A)
3

julia> A = falses(2,2);

julia> findlast(A) # returns nothing, but not printed in the REPL

julia> A = [true false; true false]
2×2 Matrix{Bool}:
 1  0
 1  0

julia> findlast(A)
CartesianIndex(2, 1)
```

source

Base.findlast – Method.

```
findlast(predicate::Function, A)
```

Return the index or key of the last element of `A` for which `predicate` returns `true`. Return `nothing` if there is no such element.

Indices or keys are of the same type as those returned by `keys(A)` and `pairs(A)`.

Examples

```
julia> A = [1, 2, 3, 4]
4-element Vector{Int64}:
 1
 2
 3
 4

julia> findlast(isodd, A)
3

julia> findlast(x -> x > 5, A) # returns nothing, but not printed in the REPL

julia> A = [1 2; 3 4]
2×2 Matrix{Int64}:
 1  2
 3  4

julia> findlast(isodd, A)
CartesianIndex(2, 1)
```

source

`Base.findnext` – Method.

```
findnext(A, i)
```

Find the next index after or including `i` of a `true` element of `A`, or `nothing` if not found.

Indices are of the same type as those returned by `keys(A)` and `pairs(A)`.

Examples

```
julia> A = [false, false, true, false]
4-element Vector{Bool}:
 0
 0
 1
 0

julia> findnext(A, 1)
3

julia> findnext(A, 4) # returns nothing, but not printed in the REPL

julia> A = [false false; true false]
2×2 Matrix{Bool}:
 0  0
 1  0
```

```
julia> findnext(A, CartesianIndex(1, 1))
CartesianIndex(2, 1)
```

source

Base.findnext – Method.

```
findnext(predicate::Function, A, i)
```

Find the next index after or including `i` of an element of `A` for which `predicate` returns `true`, or `nothing` if not found. This works for Arrays, Strings, and most other collections that support `getindex`, `keys(A)`, and `nextind`.

Indices are of the same type as those returned by `keys(A)` and `pairs(A)`.

Examples

```
julia> A = [1, 4, 2, 2];

julia> findnext(isodd, A, 1)
1

julia> findnext(isodd, A, 2) # returns nothing, but not printed in the REPL

julia> A = [1 4; 2 2];

julia> findnext(isodd, A, CartesianIndex(1, 1))
CartesianIndex(1, 1)

julia> findnext(isspace, "a b c", 3)
4
```

source

Base.findprev – Method.

```
findprev(A, i)
```

Find the previous index before or including `i` of a `true` element of `A`, or `nothing` if not found.

Indices are of the same type as those returned by `keys(A)` and `pairs(A)`.

See also: `findnext`, `findfirst`, `findall`.

Examples

```
julia> A = [false, false, true, true]
4-element Vector{Bool}:
 0
 0
 1
 1

julia> findprev(A, 3)
```

```
3

julia> findprev(A, 1) # returns nothing, but not printed in the REPL

julia> A = [false false; true true]
2×2 Matrix{Bool}:
 0  0
 1  1

julia> findprev(A, CartesianIndex(2, 1))
CartesianIndex(2, 1)
```

source

Base.findprev – Method.

```
findprev(predicate::Function, A, i)
```

Find the previous index before or including `i` of an element of `A` for which `predicate` returns `true`, or `nothing` if not found. This works for Arrays, Strings, and most other collections that support `getindex`, `keys(A)`, and `nextind`.

Indices are of the same type as those returned by `keys(A)` and `pairs(A)`.

Examples

```
julia> A = [4, 6, 1, 2]
4-element Vector{Int64}:
 4
 6
 1
 2

julia> findprev(isodd, A, 1) # returns nothing, but not printed in the REPL

julia> findprev(isodd, A, 3)
3

julia> A = [4 6; 1 2]
2×2 Matrix{Int64}:
 4  6
 1  2

julia> findprev(isodd, A, CartesianIndex(1, 2))
CartesianIndex(2, 1)

julia> findprev(isspace, "a b c", 3)
2
```

source

Base.permutedims – Function.

```
permutedims(v::AbstractVector)
```

Reshape vector v into a `1 × length(v)` row matrix. Differs from `transpose` in that the operation is not recursive, which is especially useful for arrays of non-numeric values (where the recursive `transpose` might throw an error).

Examples

Unlike `transpose`, `permutedims` can be used on vectors of arbitrary non-numeric elements, such as strings:

```
julia> permutedims(["a", "b", "c"])
1×3 Matrix{String}:
 "a"  "b"  "c"
```

For vectors of numbers, `permutedims(v)` works much like `transpose(v)` except that the return type differs (it uses `reshape` rather than a `LinearAlgebra.Transpose` view, though both share memory with the original array v):

```
julia> v = [1, 2, 3, 4]
4-element Vector{Int64}:
 1
 2
 3
 4

julia> p = permutedims(v)
1×4 Matrix{Int64}:
 1  2  3  4

julia> r = transpose(v)
1×4 transpose(::Vector{Int64}) with eltype Int64:
 1  2  3  4

julia> p == r
true

julia> typeof(r)
Transpose{Int64, Vector{Int64}}

julia> p[1] = 5; r[2] = 6; # mutating p or r also changes v

julia> v # shares memory with both p and r
4-element Vector{Int64}:
 5
 6
 3
 4
```

However, `permutedims` produces results that differ from `transpose` for vectors whose elements are themselves numeric matrices:

```
julia> V = [[[1 2; 3 4]]; [[5 6; 7 8]]]
2-element Vector{Matrix{Int64}}:
 [1 2; 3 4]
```

```
 [5 6; 7 8]

julia> permutedims(V)
1×2 Matrix{Matrix{Int64}}:
 [1 2; 3 4]  [5 6; 7 8]

julia> transpose(V)
1×2 transpose(::Vector{Matrix{Int64}}) with eltype Transpose{Int64, Matrix{Int64}}:
 [1 3; 2 4]  [5 7; 6 8]
```

source

```
permutedims(A::AbstractArray, perm)
permutedims(A::AbstractMatrix)
```

Permute the dimensions (axes) of array `A`. `perm` is a tuple or vector of `ndims(A)` integers specifying the permutation.

If `A` is a 2d array (`AbstractMatrix`), then `perm` defaults to `(2,1)`, swapping the two axes of `A` (the rows and columns of the matrix). This differs from `transpose` in that the operation is not recursive, which is especially useful for arrays of non-numeric values (where the recursive `transpose` would throw an error) and/or 2d arrays that do not represent linear operators.

For 1d arrays, see `permutedims(v::AbstractVector)`, which returns a 1-row "matrix".

See also `permutedims!`, `PermutedDimsArray`, `transpose`, `invperm`.

Examples

2d arrays:

Unlike `transpose`, `permutedims` can be used to swap rows and columns of 2d arrays of arbitrary non-numeric elements, such as strings:

```
julia> A = ["a" "b" "c"
            "d" "e" "f"]
2×3 Matrix{String}:
 "a"  "b"  "c"
 "d"  "e"  "f"

julia> permutedims(A)
3×2 Matrix{String}:
 "a"  "d"
 "b"  "e"
 "c"  "f"
```

And `permutedims` produces results that differ from `transpose` for matrices whose elements are themselves numeric matrices:

```
julia> a = [1 2; 3 4];

julia> b = [5 6; 7 8];

julia> c = [9 10; 11 12];
```

```
julia> d = [13 14; 15 16];

julia> X = [[a] [b]; [c] [d]]
2×2 Matrix{Matrix{Int64}}:
 [1 2; 3 4]     [5 6; 7 8]
 [9 10; 11 12]  [13 14; 15 16]

julia> permutedims(X)
2×2 Matrix{Matrix{Int64}}:
 [1 2; 3 4]  [9 10; 11 12]
 [5 6; 7 8]  [13 14; 15 16]

julia> transpose(X)
2×2 transpose(::Matrix{Matrix{Int64}}) with eltype Transpose{Int64, Matrix{Int64}}:
 [1 3; 2 4]  [9 11; 10 12]
 [5 7; 6 8]  [13 15; 14 16]
```

Multi-dimensional arrays

```
julia> A = reshape(Vector(1:8), (2,2,2))
2×2×2 Array{Int64, 3}:
[:, :, 1] =
 1  3
 2  4

[:, :, 2] =
 5  7
 6  8

julia> perm = (3, 1, 2); # put the last dimension first

julia> B = permutedims(A, perm)
2×2×2 Array{Int64, 3}:
[:, :, 1] =
 1  2
 5  6

[:, :, 2] =
 3  4
 7  8

julia> A == permutedims(B, invperm(perm)) # the inverse permutation
true
```

For each dimension `i` of `B = permutedims(A, perm)`, its corresponding dimension of `A` will be `perm[i]`. This means the equality `size(B, i) == size(A, perm[i])` holds.

```
julia> A = randn(5, 7, 11, 13);

julia> perm = [4, 1, 3, 2];

julia> B = permutedims(A, perm);
```

```
julia> size(B)
(13, 5, 11, 7)

julia> size(A)[perm] == ans
true
```

source

Base.permutedims! – Function.

```
permutedims!(dest, src, perm)
```

Permute the dimensions of array src and store the result in the array dest. perm is a vector specifying a permutation of length ndims(src). The preallocated array dest should have size(dest) == size(src)[perm] and is completely overwritten. No in-place permutation is supported and unexpected results will happen if src and dest have overlapping memory regions.

See also permutedims.

source

Base.PermutedDimsArrays.PermutedDimsArray – Type.

```
PermutedDimsArray(A, perm) -> B
```

Given an AbstractArray A, create a view B such that the dimensions appear to be permuted. Similar to permutedims, except that no copying occurs (B shares storage with A).

See also permutedims, invperm.

Examples

```
julia> A = rand(3,5,4);

julia> B = PermutedDimsArray(A, (3,1,2));

julia> size(B)
(4, 3, 5)

julia> B[3,1,2] == A[1,2,3]
true
```

source

Base.promote_shape – Function.

```
promote_shape(s1, s2)
```

Check two array shapes for compatibility, allowing trailing singleton dimensions, and return whichever shape has more dimensions.

Examples

```
julia> a = fill(1, (3,4,1,1,1));

julia> b = fill(1, (3,4));

julia> promote_shape(a,b)
(Base.OneTo(3), Base.OneTo(4), Base.OneTo(1), Base.OneTo(1), Base.OneTo(1))

julia> promote_shape((2,3,1,4), (2, 3, 1, 4, 1))
(2, 3, 1, 4, 1)
```

source

49.7 Array functions

Base.accumulate – Function.

```
accumulate(op, A; dims::Integer, [init])
```

Cumulative operation op along the dimension dims of A (providing dims is optional for vectors). An initial value init may optionally be provided by a keyword argument. See also accumulate! to use a preallocated output array, both for performance and to control the precision of the output (e.g. to avoid overflow).

For common operations there are specialized variants of accumulate, see cumsum, cumprod. For a lazy version, see Iterators.accumulate.

Julia 1.5

accumulate on a non-array iterator requires at least Julia 1.5.

Examples

```
julia> accumulate(+, [1,2,3])
3-element Vector{Int64}:
 1
 3
 6

julia> accumulate(min, (1, -2, 3, -4, 5), init=0)
(0, -2, -2, -4, -4)

julia> accumulate(/, (2, 4, Inf), init=100)
(50.0, 12.5, 0.0)

julia> accumulate(=>, i^2 for i in 1:3)
3-element Vector{Any}:
          1
       1 => 4
 (1 => 4) => 9

julia> accumulate(+, fill(1, 3, 4))
3×4 Matrix{Int64}:
 1  4  7  10
```

```
 2  5  8  11
 3  6  9  12

julia> accumulate(+, fill(1, 2, 5), dims=2, init=100.0)
2×5 Matrix{Float64}:
 101.0  102.0  103.0  104.0  105.0
 101.0  102.0  103.0  104.0  105.0
```

source

Base.accumulate! – Function.

```
accumulate!(op, B, A; [dims], [init])
```

Cumulative operation op on A along the dimension dims, storing the result in B. Providing dims is optional for vectors. If the keyword argument init is given, its value is used to instantiate the accumulation.

Warning

Behavior can be unexpected when any mutated argument shares memory with any other argument.

See also accumulate, cumsum!, cumprod!.

Examples

```
julia> x = [1, 0, 2, 0, 3];

julia> y = rand(5);

julia> accumulate!(+, y, x);

julia> y
5-element Vector{Float64}:
 1.0
 1.0
 3.0
 3.0
 6.0

julia> A = [1 2 3; 4 5 6];

julia> B = similar(A);

julia> accumulate!(-, B, A, dims=1)
2×3 Matrix{Int64}:
  1   2   3
 -3  -3  -3

julia> accumulate!(*, B, A, dims=2, init=10)
2×3 Matrix{Int64}:
 10   20    60
 40  200  1200
```

source

Base.cumprod – Function.

```
cumprod(itr)
```

Cumulative product of an iterator.

See also cumprod!, accumulate, cumsum.

Julia 1.5

cumprod on a non-array iterator requires at least Julia 1.5.

Examples

```
julia> cumprod(fill(1//2, 3))
3-element Vector{Rational{Int64}}:
 1//2
 1//4
 1//8

julia> cumprod((1, 2, 1, 3, 1))
(1, 2, 2, 6, 6)

julia> cumprod("julia")
5-element Vector{String}:
 "j"
 "ju"
 "jul"
 "juli"
 "julia"
```

source

```
cumprod(A; dims::Integer)
```

Cumulative product along the dimension dim. See also cumprod! to use a preallocated output array, both for performance and to control the precision of the output (e.g. to avoid overflow).

Examples

```
julia> a = Int8[1 2 3; 4 5 6];

julia> cumprod(a, dims=1)
2×3 Matrix{Int64}:
 1   2   3
 4  10  18

julia> cumprod(a, dims=2)
2×3 Matrix{Int64}:
 1   2    6
 4  20  120
```

source

Base.cumprod! – Function.

```
cumprod!(y::AbstractVector, x::AbstractVector)
```

Cumulative product of a vector x, storing the result in y. See also cumprod.

Warning

Behavior can be unexpected when any mutated argument shares memory with any other argument.

source

```
cumprod!(B, A; dims::Integer)
```

Cumulative product of A along the dimension dims, storing the result in B. See also cumprod.

Warning

Behavior can be unexpected when any mutated argument shares memory with any other argument.

source

Base.cumsum – Function.

```
cumsum(itr)
```

Cumulative sum of an iterator.

See also accumulate to apply functions other than +.

Julia 1.5

cumsum on a non-array iterator requires at least Julia 1.5.

Examples

```
julia> cumsum(1:3)
3-element Vector{Int64}:
 1
 3
 6

julia> cumsum((true, false, true, false, true))
(1, 1, 2, 2, 3)

julia> cumsum(fill(1, 2) for i in 1:3)
3-element Vector{Vector{Int64}}:
 [1, 1]
 [2, 2]
 [3, 3]
```

source

```
cumsum(A; dims::Integer)
```

Cumulative sum along the dimension `dims`. See also `cumsum!` to use a preallocated output array, both for performance and to control the precision of the output (e.g. to avoid overflow).

Examples

```
julia> a = [1 2 3; 4 5 6]
2×3 Matrix{Int64}:
 1  2  3
 4  5  6

julia> cumsum(a, dims=1)
2×3 Matrix{Int64}:
 1  2  3
 5  7  9

julia> cumsum(a, dims=2)
2×3 Matrix{Int64}:
 1  3   6
 4  9  15
```

Note

The return array's `eltype` is `Int` for signed integers of less than system word size and `UInt` for unsigned integers of less than system word size. To preserve `eltype` of arrays with small signed or unsigned integer `accumulate(+, A)` should be used.

```
julia> cumsum(Int8[100, 28])
2-element Vector{Int64}:
 100
 128

julia> accumulate(+,Int8[100, 28])
2-element Vector{Int8}:
  100
 -128
```

In the former case, the integers are widened to system word size and therefore the result is `Int64[100, 128]`. In the latter case, no such widening happens and integer overflow results in `Int8[100, -128]`.

source

`Base.cumsum!` – Function.

```
cumsum!(B, A; dims::Integer)
```

Cumulative sum of `A` along the dimension `dims`, storing the result in `B`. See also `cumsum`.

Warning

Behavior can be unexpected when any mutated argument shares memory with any other argument.

source

Base.diff – Function.

```
diff(A::AbstractVector)
diff(A::AbstractArray; dims::Integer)
```

Finite difference operator on a vector or a multidimensional array A. In the latter case the dimension to operate on needs to be specified with the dims keyword argument.

Julia 1.1

diff for arrays with dimension higher than 2 requires at least Julia 1.1.

Examples

```
julia> a = [2 4; 6 16]
2×2 Matrix{Int64}:
 2   4
 6  16

julia> diff(a, dims=2)
2×1 Matrix{Int64}:
  2
 10

julia> diff(vec(a))
3-element Vector{Int64}:
  4
 -2
 12
```

source

Base.repeat – Function.

```
repeat(A::AbstractArray; inner=ntuple(Returns(1), ndims(A)), outer=ntuple(Returns(1),
↪  ndims(A)))
```

Construct an array by repeating the entries of A. The i-th element of inner specifies the number of times that the individual entries of the i-th dimension of A should be repeated. The i-th element of outer specifies the number of times that a slice along the i-th dimension of A should be repeated. If inner or outer are omitted, no repetition is performed.

Examples

```
julia> repeat(1:2, inner=2)
4-element Vector{Int64}:
 1
```

```
 1
 2
 2

julia> repeat(1:2, outer=2)
4-element Vector{Int64}:
 1
 2
 1
 2

julia> repeat([1 2; 3 4], inner=(2, 1), outer=(1, 3))
4×6 Matrix{Int64}:
 1  2  1  2  1  2
 1  2  1  2  1  2
 3  4  3  4  3  4
 3  4  3  4  3  4
```

source

```
repeat(A::AbstractArray, counts::Integer...)
```

Construct an array by repeating array `A` a given number of times in each dimension, specified by `counts`.

See also: `fill`, `Iterators.repeated`, `Iterators.cycle`.

Examples

```
julia> repeat([1, 2, 3], 2)
6-element Vector{Int64}:
 1
 2
 3
 1
 2
 3

julia> repeat([1, 2, 3], 2, 3)
6×3 Matrix{Int64}:
 1  1  1
 2  2  2
 3  3  3
 1  1  1
 2  2  2
 3  3  3
```

source

```
repeat(c::AbstractChar, r::Integer)::String
```

Repeat a character `r` times. This can equivalently be accomplished by calling `c^r`.

Examples

```
julia> repeat('A', 3)
"AAA"
```

source

```
repeat(s::AbstractString, r::Integer)
```

Repeat a string `r` times. This can be written as `s^r`.

See also `^`.

Examples

```
julia> repeat("ha", 3)
"hahaha"
```

source

`Base.rot180` – Function.

```
rot180(A, k)
```

Rotate matrix `A` 180 degrees an integer `k` number of times. If `k` is even, this is equivalent to a `copy`.

Examples

```
julia> a = [1 2; 3 4]
2×2 Matrix{Int64}:
 1  2
 3  4

julia> rot180(a,1)
2×2 Matrix{Int64}:
 4  3
 2  1

julia> rot180(a,2)
2×2 Matrix{Int64}:
 1  2
 3  4
```

source

```
rot180(A)
```

Rotate matrix `A` 180 degrees.

Examples

```
julia> a = [1 2; 3 4]
2×2 Matrix{Int64}:
 1  2
 3  4
```

```
julia> rot180(a)
2×2 Matrix{Int64}:
 4  3
 2  1
```

source

Base.rotl90 – Function.

```
rotl90(A, k)
```

Left-rotate matrix A 90 degrees counterclockwise an integer k number of times. If k is a multiple of four (including zero), this is equivalent to a copy.

Examples

```
julia> a = [1 2; 3 4]
2×2 Matrix{Int64}:
 1  2
 3  4

julia> rotl90(a,1)
2×2 Matrix{Int64}:
 2  4
 1  3

julia> rotl90(a,2)
2×2 Matrix{Int64}:
 4  3
 2  1

julia> rotl90(a,3)
2×2 Matrix{Int64}:
 3  1
 4  2

julia> rotl90(a,4)
2×2 Matrix{Int64}:
 1  2
 3  4
```

source

```
rotl90(A)
```

Rotate matrix A left 90 degrees.

Examples

```
julia> a = [1 2; 3 4]
2×2 Matrix{Int64}:
 1  2
 3  4
```

```
julia> rotl90(a)
2×2 Matrix{Int64}:
 2  4
 1  3
```

source

Base.rotr90 – Function.

```
rotr90(A, k)
```

Right-rotate matrix A 90 degrees clockwise an integer k number of times. If k is a multiple of four (including zero), this is equivalent to a copy.

Examples

```
julia> a = [1 2; 3 4]
2×2 Matrix{Int64}:
 1  2
 3  4

julia> rotr90(a,1)
2×2 Matrix{Int64}:
 3  1
 4  2

julia> rotr90(a,2)
2×2 Matrix{Int64}:
 4  3
 2  1

julia> rotr90(a,3)
2×2 Matrix{Int64}:
 2  4
 1  3

julia> rotr90(a,4)
2×2 Matrix{Int64}:
 1  2
 3  4
```

source

```
rotr90(A)
```

Rotate matrix A right 90 degrees.

Examples

```
julia> a = [1 2; 3 4]
2×2 Matrix{Int64}:
 1  2
 3  4
```

```
julia> rotr90(a)
2×2 Matrix{Int64}:
 3  1
 4  2
```

source

Base.mapslices – Function.

```
mapslices(f, A; dims)
```

Transform the given dimensions of array A by applying a function f on each slice of the form A[..., :, ..., :, ...], with a colon at each d in dims. The results are concatenated along the remaining dimensions.

For example, if dims = [1,2] and A is 4-dimensional, then f is called on x = A[:,:,i,j] for all i and j, and f(x) becomes R[:,:,i,j] in the result R.

See also eachcol or eachslice, used with map or stack.

Examples

```
julia> A = reshape(1:30,(2,5,3))
2×5×3 reshape(::UnitRange{Int64}, 2, 5, 3) with eltype Int64:
[:, :, 1] =
 1  3  5  7   9
 2  4  6  8  10

[:, :, 2] =
 11  13  15  17  19
 12  14  16  18  20

[:, :, 3] =
 21  23  25  27  29
 22  24  26  28  30

julia> f(x::Matrix) = fill(x[1,1], 1,4);  # returns a 1×4 matrix

julia> B = mapslices(f, A, dims=(1,2))
1×4×3 Array{Int64, 3}:
[:, :, 1] =
 1  1  1  1

[:, :, 2] =
 11  11  11  11

[:, :, 3] =
 21  21  21  21

julia> f2(x::AbstractMatrix) = fill(x[1,1], 1,4);

julia> B == stack(f2, eachslice(A, dims=3))
true
```

```
julia> g(x) = x[begin] // x[end-1];  # returns a number

julia> mapslices(g, A, dims=[1,3])
1×5×1 Array{Rational{Int64}, 3}:
[:, :, 1] =
 1//21  3//23  1//5  7//27  9//29

julia> map(g, eachslice(A, dims=2))
5-element Vector{Rational{Int64}}:
 1//21
 3//23
 1//5
 7//27
 9//29

julia> mapslices(sum, A; dims=(1,3)) == sum(A; dims=(1,3))
true
```

Notice that in `eachslice(A; dims=2)`, the specified dimension is the one *without* a colon in the slice. This is `view(A,:,i,:)`, whereas `mapslices(f, A; dims=(1,3))` uses `A[:,i,:]`. The function `f` may mutate values in the slice without affecting `A`.

source

`Base.eachrow` – Function.

```
eachrow(A::AbstractVecOrMat) <: AbstractVector
```

Create a `RowSlices` object that is a vector of rows of matrix or vector `A`. Row slices are returned as `AbstractVector` views of `A`.

For the inverse, see `stack(rows; dims=1)`.

See also `eachcol`, `eachslice` and `mapslices`.

Julia 1.1

This function requires at least Julia 1.1.

Julia 1.9

Prior to Julia 1.9, this returned an iterator.

Examples

```
julia> a = [1 2; 3 4]
2×2 Matrix{Int64}:
 1  2
 3  4

julia> s = eachrow(a)
2-element RowSlices{Matrix{Int64}, Tuple{Base.OneTo{Int64}}, SubArray{Int64, 1, Matrix{Int64},
↪  Tuple{Int64, Base.Slice{Base.OneTo{Int64}}}, true}}:
 [1, 2]
```

```
 [3, 4]

julia> s[1]
2-element view(::Matrix{Int64}, 1, :) with eltype Int64:
 1
 2
```

source

Base.eachcol – Function.

```
eachcol(A::AbstractVecOrMat) <: AbstractVector
```

Create a ColumnSlices object that is a vector of columns of matrix or vector A. Column slices are returned as AbstractVector views of A.

For the inverse, see stack(cols) or reduce(hcat, cols).

See also eachrow, eachslice and mapslices.

Julia 1.1

This function requires at least Julia 1.1.

Julia 1.9

Prior to Julia 1.9, this returned an iterator.

Examples

```
julia> a = [1 2; 3 4]
2×2 Matrix{Int64}:
 1  2
 3  4

julia> s = eachcol(a)
2-element ColumnSlices{Matrix{Int64}, Tuple{Base.OneTo{Int64}}, SubArray{Int64, 1,
↪  Matrix{Int64}, Tuple{Base.Slice{Base.OneTo{Int64}}, Int64}, true}}:
 [1, 3]
 [2, 4]

julia> s[1]
2-element view(::Matrix{Int64}, :, 1) with eltype Int64:
 1
 3
```

source

Base.eachslice – Function.

```
eachslice(A::AbstractArray; dims, drop=true)
```

Create a sliced object, usually Slices, that is an array of slices over dimensions dims of A, returning views that select all the data from the other dimensions in A. dims can either be an integer or a tuple of integers.

If `drop = true` (the default), the outer slices will drop the inner dimensions, and the ordering of the dimensions will match those in `dims`. If `drop = false`, then the slices object will have the same dimensionality as the underlying array, with inner dimensions having size 1.

See `stack(slices; dims)` for the inverse of `eachslice(A; dims::Integer)`.

See also `eachrow`, `eachcol`, `mapslices` and `selectdim`.

Julia 1.1

This function requires at least Julia 1.1.

Julia 1.9

Prior to Julia 1.9, this returned an iterator, and only a single dimension `dims` was supported.

Examples

```
julia> m = [1 2 3; 4 5 6; 7 8 9]
3×3 Matrix{Int64}:
 1  2  3
 4  5  6
 7  8  9

julia> s = eachslice(m, dims=1)
3-element RowSlices{Matrix{Int64}, Tuple{Base.OneTo{Int64}}, SubArray{Int64, 1, Matrix{Int64},
↪  Tuple{Int64, Base.Slice{Base.OneTo{Int64}}}, true}}:
 [1, 2, 3]
 [4, 5, 6]
 [7, 8, 9]

julia> s[1]
3-element view(::Matrix{Int64}, 1, :) with eltype Int64:
 1
 2
 3

julia> eachslice(m, dims=1, drop=false)
3×1 Slices{Matrix{Int64}, Tuple{Int64, Colon}, Tuple{Base.OneTo{Int64}, Base.OneTo{Int64}},
↪  SubArray{Int64, 1, Matrix{Int64}, Tuple{Int64, Base.Slice{Base.OneTo{Int64}}}, true}, 2}:
 [1, 2, 3]
 [4, 5, 6]
 [7, 8, 9]
```

source

49.8 Combinatorics

`Base.invperm` – Function.

```
invperm(v)
```

Return the inverse permutation of `v`. If `B = A[v]`, then `A == B[invperm(v)]`.

See also sortperm, invpermute!, isperm, permutedims.

Examples

```
julia> p = (2, 3, 1);

julia> invperm(p)
(3, 1, 2)

julia> v = [2; 4; 3; 1];

julia> invperm(v)
4-element Vector{Int64}:
 4
 1
 3
 2

julia> A = ['a','b','c','d'];

julia> B = A[v]
4-element Vector{Char}:
 'b': ASCII/Unicode U+0062 (category Ll: Letter, lowercase)
 'd': ASCII/Unicode U+0064 (category Ll: Letter, lowercase)
 'c': ASCII/Unicode U+0063 (category Ll: Letter, lowercase)
 'a': ASCII/Unicode U+0061 (category Ll: Letter, lowercase)

julia> B[invperm(v)]
4-element Vector{Char}:
 'a': ASCII/Unicode U+0061 (category Ll: Letter, lowercase)
 'b': ASCII/Unicode U+0062 (category Ll: Letter, lowercase)
 'c': ASCII/Unicode U+0063 (category Ll: Letter, lowercase)
 'd': ASCII/Unicode U+0064 (category Ll: Letter, lowercase)
```

source

Base.isperm – Function.

```
isperm(v)::Bool
```

Return true if v is a valid permutation.

Examples

```
julia> isperm([1; 2])
true

julia> isperm([1; 3])
false
```

source

Base.permute! – Method.

```
permute!(v, p)
```

Permute vector v according to permutation p, storing the result back into v. No checking is done to verify that p is a permutation.

To return a new permutation, use `v[p]`. This is generally faster than `permute!(v, p)`; it is even faster to write into a pre-allocated output array with `u .= @view v[p]`. (Even though `permute!` overwrites v in-place, it internally requires some allocation.)

Warning

Behavior can be unexpected when any mutated argument shares memory with any other argument.

See also `invpermute!`.

Examples

```
julia> A = [1, 1, 3, 4];

julia> perm = [2, 4, 3, 1];

julia> permute!(A, perm);

julia> A
4-element Vector{Int64}:
 1
 4
 3
 1
```

source

`Base.invpermute!` – Function.

```
invpermute!(v, p)
```

Like `permute!`, but the inverse of the given permutation is applied.

Note that if you have a pre-allocated output array (e.g. `u = similar(v)`), it is quicker to instead employ `u[p] = v`. (`invpermute!` internally allocates a copy of the data.)

Warning

Behavior can be unexpected when any mutated argument shares memory with any other argument.

Examples

```
julia> A = [1, 1, 3, 4];

julia> perm = [2, 4, 3, 1];

julia> invpermute!(A, perm);

julia> A
```

```
4-element Vector{Int64}:
 4
 1
 3
 1
```

source

Base.reverse – Method.

```
reverse(A; dims=:)
```

Reverse A along dimension dims, which can be an integer (a single dimension), a tuple of integers (a tuple of dimensions) or : (reverse along all the dimensions, the default). See also reverse! for in-place reversal.

Examples

```
julia> b = Int64[1 2; 3 4]
2×2 Matrix{Int64}:
 1  2
 3  4

julia> reverse(b, dims=2)
2×2 Matrix{Int64}:
 2  1
 4  3

julia> reverse(b)
2×2 Matrix{Int64}:
 4  3
 2  1
```

Julia 1.6

Prior to Julia 1.6, only single-integer dims are supported in reverse.

source

Base.reverseind – Function.

```
reverseind(v, i)
```

Given an index i in reverse(v), return the corresponding index in v so that v[reverseind(v,i)] == reverse(v)[i]. (This can be nontrivial in cases where v contains non-ASCII characters.)

Examples

```
julia> s = "Julia□"
"Julia□"

julia> r = reverse(s)
"□ailuJ"
```

```
julia> for i in eachindex(s)
           print(r[reverseind(r, i)])
       end
Julia
```

source

Base.reverse! – Function.

```
reverse!(A; dims=:)
```

Like reverse, but operates in-place in A.

Julia 1.6

Multidimensional reverse! requires Julia 1.6.

source

```
reverse!(v [, start=firstindex(v) [, stop=lastindex(v) ]]) -> v
```

In-place version of reverse.

Examples

```
julia> A = Vector(1:5)
5-element Vector{Int64}:
 1
 2
 3
 4
 5

julia> reverse!(A);

julia> A
5-element Vector{Int64}:
 5
 4
 3
 2
 1
```

source

Chapter 50

Tasks

`Core.Task` – Type.

```
Task(func[, reserved_stack::Int])
```

Create a `Task` (i.e. coroutine) to execute the given function `func` (which must be callable with no arguments). The task exits when this function returns. The task will run in the "world age" from the parent at construction when `schedule`d.

The optional `reserved_stack` argument specifies the size of the stack available for this task, in bytes. The default, `0`, uses the system-dependent stack size default.

Warning

By default tasks will have the sticky bit set to true `t.sticky`. This models the historic default for `@async`. Sticky tasks can only be run on the worker thread they are first scheduled on, and when scheduled will make the task that they were scheduled from sticky. To obtain the behavior of `Threads.@spawn` set the sticky bit manually to `false`.

Examples

```
julia> a() = sum(i for i in 1:1000);

julia> b = Task(a);
```

In this example, `b` is a runnable `Task` that hasn't started yet.

source

`Base.@task` – Macro.

```
@task
```

Wrap an expression in a `Task` without executing it, and return the `Task`. This only creates a task, and does not run it.

Warning

By default tasks will have the sticky bit set to true `t.sticky`. This models the historic default for `@async`. Sticky tasks can only be run on the worker thread they are first scheduled on, and when scheduled will make the task that they were scheduled from sticky. To obtain the behavior of `Threads.@spawn` set the sticky bit manually to `false`.

Examples

```
julia> a1() = sum(i for i in 1:1000);

julia> b = @task a1();

julia> istaskstarted(b)
false

julia> schedule(b);

julia> yield();

julia> istaskdone(b)
true
```

source

Base.@async – Macro.

```
@async
```

Wrap an expression in a `Task` and add it to the local machine's scheduler queue.

Values can be interpolated into `@async` via `$`, which copies the value directly into the constructed underlying closure. This allows you to insert the *value* of a variable, isolating the asynchronous code from changes to the variable's value in the current task.

Warning

It is strongly encouraged to favor `Threads.@spawn` over `@async` always **even when no parallelism is required** especially in publicly distributed libraries. This is because a use of `@async` disables the migration of the *parent* task across worker threads in the current implementation of Julia. Thus, seemingly innocent use of `@async` in a library function can have a large impact on the performance of very different parts of user applications.

Julia 1.4

Interpolating values via `$` is available as of Julia 1.4.

source

Base.asyncmap – Function.

```
asyncmap(f, c...; ntasks=0, batch_size=nothing)
```

Uses multiple concurrent tasks to map `f` over a collection (or multiple equal length collections). For multiple collection arguments, `f` is applied elementwise.

The output is guaranteed to be the same order as the elements of the collection(s) `c`.

`ntasks` specifies the number of tasks to run concurrently. Depending on the length of the collections, if `ntasks` is unspecified, up to 100 tasks will be used for concurrent mapping.

`ntasks` can also be specified as a zero-arg function. In this case, the number of tasks to run in parallel is checked before processing every element and a new task started if the value of `ntasks_func` is greater than the current number of tasks.

If `batch_size` is specified, the collection is processed in batch mode. `f` must then be a function that must accept a `Vector` of argument tuples and must return a vector of results. The input vector will have a length of `batch_size` or less.

The following examples highlight execution in different tasks by returning the `objectid` of the tasks in which the mapping function is executed.

First, with `ntasks` undefined, each element is processed in a different task.

```
julia> tskoid() = objectid(current_task());

julia> asyncmap(x->tskoid(), 1:5)
5-element Vector{UInt64}:
 0x6e15e66c75c75853
 0x440f8819a1baa682
 0x9fb3eeadd0c83985
 0xebd3e35fe90d4050
 0x29efc93edce2b961

julia> length(unique(asyncmap(x->tskoid(), 1:5)))
5
```

With `ntasks=2` all elements are processed in 2 tasks.

```
julia> asyncmap(x->tskoid(), 1:5; ntasks=2)
5-element Vector{UInt64}:
 0x027ab1680df7ae94
 0xa23d2f80cd7cf157
 0x027ab1680df7ae94
 0xa23d2f80cd7cf157
 0x027ab1680df7ae94

julia> length(unique(asyncmap(x->tskoid(), 1:5; ntasks=2)))
2
```

With `batch_size` defined, the mapping function needs to be changed to accept an array of argument tuples and return an array of results. `map` is used in the modified mapping function to achieve this.

```
julia> batch_func(input) = map(x->string("args_tuple: ", x, ", element_val: ", x[1], ", task:
↪  ", tskoid()), input)
batch_func (generic function with 1 method)
```

```
julia> asyncmap(batch_func, 1:5; ntasks=2, batch_size=2)
5-element Vector{String}:
 "args_tuple: (1,), element_val: 1, task: 9118321258196414413"
 "args_tuple: (2,), element_val: 2, task: 4904288162898683522"
 "args_tuple: (3,), element_val: 3, task: 9118321258196414413"
 "args_tuple: (4,), element_val: 4, task: 4904288162898683522"
 "args_tuple: (5,), element_val: 5, task: 9118321258196414413"
```

source

Base.asyncmap! – Function.

```
asyncmap!(f, results, c...; ntasks=0, batch_size=nothing)
```

Like asyncmap, but stores output in results rather than returning a collection.

Warning

Behavior can be unexpected when any mutated argument shares memory with any other argument.

source

Base.current_task – Function.

```
current_task()
```

Get the currently running Task.

source

Base.istaskdone – Function.

```
istaskdone(t::Task)::Bool
```

Determine whether a task has exited.

Examples

```
julia> a2() = sum(i for i in 1:1000);

julia> b = Task(a2);

julia> istaskdone(b)
false

julia> schedule(b);

julia> yield();

julia> istaskdone(b)
true
```

source

Base.istaskstarted – Function.

```
istaskstarted(t::Task)::Bool
```

Determine whether a task has started executing.

Examples

```
julia> a3() = sum(i for i in 1:1000);

julia> b = Task(a3);

julia> istaskstarted(b)
false
```

source

Base.istaskfailed – Function.

```
istaskfailed(t::Task)::Bool
```

Determine whether a task has exited because an exception was thrown.

Examples

```
julia> a4() = error("task failed");

julia> b = Task(a4);

julia> istaskfailed(b)
false

julia> schedule(b);

julia> yield();

julia> istaskfailed(b)
true
```

Julia 1.3

This function requires at least Julia 1.3.

source

Base.task_local_storage – Method.

```
task_local_storage(key)
```

Look up the value of a key in the current task's task-local storage.

source

Base.task_local_storage – Method.

```
task_local_storage(key, value)
```

Assign a value to a key in the current task's task-local storage.

source

`Base.task_local_storage` – Method.

```
task_local_storage(body, key, value)
```

Call the function body with a modified task-local storage, in which `value` is assigned to `key`; the previous value of `key`, or lack thereof, is restored afterwards. Useful for emulating dynamic scoping.

source

`Core.ConcurrencyViolationError` – Type.

```
ConcurrencyViolationError(msg) <: Exception
```

An error thrown when a detectable violation of concurrent semantics has occurred.

A non-exhaustive list of examples of when this is used include:

- Throwing when a deadlock has been detected (e.g. `wait(current_task())`)
- Known-unsafe behavior is attempted (e.g. `yield(current_task)`)
- A known non-threadsafe datastructure is attempted to be modified from multiple concurrent tasks
- A lock is being unlocked that wasn't locked by this task

source

50.1 Scheduling

`Base.yield` – Function.

```
yield(t::Task, arg = nothing)
```

A fast, unfair-scheduling version of `schedule(t, arg); yield()` which immediately yields to `t` before calling the scheduler.

Throws a `ConcurrencyViolationError` if `t` is the currently running task.

source

```
yield()
```

Switch to the scheduler to allow another scheduled task to run. A task that calls this function is still runnable, and will be restarted immediately if there are no other runnable tasks.

source

`Base.yieldto` – Function.

```
yieldto(t::Task, arg = nothing)
```

Switch to the given task. The first time a task is switched to, the task's function is called with no arguments. On subsequent switches, `arg` is returned from the task's last call to `yieldto`. This is a low-level call that only switches tasks, not considering states or scheduling in any way. Its use is discouraged.

source

Base.sleep – Function.

```
sleep(seconds)
```

Block the current task for a specified number of seconds. The minimum sleep time is 1 millisecond or input of `0.001`.

source

Base.schedule – Function.

```
schedule(t::Task, [val]; error=false)
```

Add a Task to the scheduler's queue. This causes the task to run constantly when the system is otherwise idle, unless the task performs a blocking operation such as `wait`.

If a second argument `val` is provided, it will be passed to the task (via the return value of `yieldto`) when it runs again. If `error` is `true`, the value is raised as an exception in the woken task.

Warning

It is incorrect to use `schedule` on an arbitrary `Task` that has already been started. See the API reference for more information.

Warning

By default tasks will have the sticky bit set to true `t.sticky`. This models the historic default for `@async`. Sticky tasks can only be run on the worker thread they are first scheduled on, and when scheduled will make the task that they were scheduled from sticky. To obtain the behavior of `Threads.@spawn` set the sticky bit manually to `false`.

Examples

```
julia> a5() = sum(i for i in 1:1000);

julia> b = Task(a5);

julia> istaskstarted(b)
false

julia> schedule(b);

julia> yield();

julia> istaskstarted(b)
true
```

```
julia> istaskdone(b)
true
```

source

50.2 Synchronization

Base.errormonitor – Function.

```
errormonitor(t::Task)
```

Print an error log to stderr if task t fails.

Examples

```
julia> wait(errormonitor(Threads.@spawn error("task failed")); throw = false)
Unhandled Task ERROR: task failed
Stacktrace:
[...]
```

source

Base.@sync – Macro.

```
@sync
```

Wait until all lexically-enclosed uses of @async, @spawn, Distributed.@spawnat and Distributed.@distributed are complete. All exceptions thrown by enclosed async operations are collected and thrown as a CompositeException.

Examples

```
julia> Threads.nthreads()
4

julia> @sync begin
           Threads.@spawn println("Thread-id $(Threads.threadid()), task 1")
           Threads.@spawn println("Thread-id $(Threads.threadid()), task 2")
       end;
Thread-id 3, task 1
Thread-id 1, task 2
```

source

Base.wait – Function.

```
wait([x])
```

Block the current task until some event occurs.

- Channel: Wait for a value to be appended to the channel.
- Condition: Wait for notify on a condition and return the val parameter passed to notify. See the Condition-specific docstring of wait for the exact behavior.

- `Process`: Wait for a process or process chain to exit. The `exitcode` field of a process can be used to determine success or failure.
- `Task`: Wait for a `Task` to finish. See the `Task`-specific docstring of `wait` for the exact behavior.
- `RawFD`: Wait for changes on a file descriptor (see the `FileWatching` package).

If no argument is passed, the task blocks for an undefined period. A task can only be restarted by an explicit call to `schedule` or `yieldto`.

Often `wait` is called within a `while` loop to ensure a waited-for condition is met before proceeding.

source

`Base.waitany` – Function.

```
waitany(tasks; throw=true) -> (done_tasks, remaining_tasks)
```

Wait until at least one of the given tasks have been completed.

If `throw` is `true`, throw `CompositeException` when one of the completed tasks completes with an exception.

The return value consists of two task vectors. The first one consists of completed tasks, and the other consists of uncompleted tasks.

Warning

This may scale poorly compared to writing code that uses multiple individual tasks that each runs serially, since this needs to scan the list of `tasks` each time and synchronize with each one every time this is called. Or consider using `waitall(tasks; failfast=true)` instead.

Julia 1.12

This function requires at least Julia 1.12.

source

`Base.waitall` – Function.

```
waitall(tasks; failfast=true, throw=true) -> (done_tasks, remaining_tasks)
```

Wait until all the given tasks have been completed.

If `failfast` is `true`, the function will return when at least one of the given tasks is finished by exception. If `throw` is `true`, throw `CompositeException` when one of the completed tasks has failed.

`failfast` and `throw` keyword arguments work independently; when only `throw=true` is specified, this function waits for all the tasks to complete.

The return value consists of two task vectors. The first one consists of completed tasks, and the other consists of uncompleted tasks.

Julia 1.12

This function requires at least Julia 1.12.

source

Base.fetch – Method.

```
fetch(t::Task)
```

Wait for a Task to finish, then return its result value. If the task fails with an exception, a TaskFailedException (which wraps the failed task) is thrown.

source

Base.fetch – Method.

```
fetch(x::Any)
```

Return x.

source

Base.timedwait – Function.

```
timedwait(testcb, timeout::Real; pollint::Real=0.1)
```

Wait until testcb() returns true or timeout seconds have passed, whichever is earlier. The test function is polled every pollint seconds. The minimum value for pollint is 0.001 seconds, that is, 1 millisecond.

Return :ok or :timed_out.

Examples

```
julia> cb() = (sleep(5); return);

julia> t = @async cb();

julia> timedwait(()->istaskdone(t), 1)
:timed_out

julia> timedwait(()->istaskdone(t), 6.5)
:ok
```

source

Base.Condition – Type.

```
Condition()
```

Create an edge-triggered event source that tasks can wait for. Tasks that call wait on a Condition are suspended and queued. Tasks are woken up when notify is later called on the Condition. Waiting on a condition can return a value or raise an error if the optional arguments of notify are used. Edge triggering means that only tasks waiting at the time notify is called can be woken up. For level-triggered notifications, you must keep extra state to keep track of whether a notification has happened. The Channel and Threads.Event types do this, and can be used for level-triggered events.

This object is NOT thread-safe. See Threads.Condition for a thread-safe version.

source

`Base.Threads.Condition` – Type.

```
Threads.Condition([lock])
```

A thread-safe version of `Base.Condition`.

To call `wait` or `notify` on a `Threads.Condition`, you must first call `lock` on it. When `wait` is called, the lock is atomically released during blocking, and will be reacquired before `wait` returns. Therefore idiomatic use of a `Threads.Condition` `c` looks like the following:

```
lock(c)
try
    while !thing_we_are_waiting_for
        wait(c)
    end
finally
    unlock(c)
end
```

Julia 1.2

This functionality requires at least Julia 1.2.

source

`Base.Event` – Type.

```
Event([autoreset=false])
```

Create a level-triggered event source. Tasks that call `wait` on an `Event` are suspended and queued until `notify` is called on the `Event`. After `notify` is called, the `Event` remains in a signaled state and tasks will no longer block when waiting for it, until `reset` is called.

If `autoreset` is true, at most one task will be released from `wait` for each call to `notify`.

This provides an acquire & release memory ordering on notify/wait.

Julia 1.1

This functionality requires at least Julia 1.1.

Julia 1.8

The `autoreset` functionality and memory ordering guarantee requires at least Julia 1.8.

source

`Base.notify` – Function.

```
notify(condition, val=nothing; all=true, error=false)
```

Wake up tasks waiting for a condition, passing them `val`. If `all` is `true` (the default), all waiting tasks are woken, otherwise only one is. If `error` is `true`, the passed value is raised as an exception in the woken tasks.

Return the count of tasks woken up. Return 0 if no tasks are waiting on `condition`.

source

`Base.reset` – Method.

```
reset(::Event)
```

Reset an `Event` back into an un-set state. Then any future calls to `wait` will block until `notify` is called again.

source

`Base.Semaphore` – Type.

```
Semaphore(sem_size)
```

Create a counting semaphore that allows at most `sem_size` acquires to be in use at any time. Each acquire must be matched with a release.

This provides a acquire & release memory ordering on acquire/release calls.

source

`Base.acquire` – Function.

```
acquire(f, s::Semaphore)
```

Execute `f` after acquiring from Semaphore `s`, and `release` on completion or error.

For example, a do-block form that ensures only 2 calls of `foo` will be active at the same time:

```
s = Base.Semaphore(2)
@sync for _ in 1:100
    Threads.@spawn begin
        Base.acquire(s) do
            foo()
        end
    end
end
```

Julia 1.8

This method requires at least Julia 1.8.

source

```
acquire(s::Semaphore)
```

Wait for one of the `sem_size` permits to be available, blocking until one can be acquired.

source

`Base.@acquire` – Macro.

```
Base.@acquire s::Semaphore expr
```

Macro version of `Base.acquire(f, s::Semaphore)` but with `expr` instead of `f` function. Expands to:

```
Base.acquire(s)
try
    expr
finally
    Base.release(s)
end
```

This is similar to using `acquire` with a do block, but avoids creating a closure.

Julia 1.13

`Base.@acquire` was added in Julia 1.13

source

`Base.release` – Function.

```
release(s::Semaphore)
```

Return one permit to the pool, possibly allowing another task to acquire it and resume execution.

source

`Base.AbstractLock` – Type.

```
AbstractLock
```

Abstract supertype describing types that implement the synchronization primitives: `lock`, `trylock`, `unlock`, and `islocked`.

source

`Base.lock` – Function.

```
lock(f::Function, l::Lockable)
```

Acquire the lock associated with `l`, execute `f` with the lock held, and release the lock when `f` returns. `f` will receive one positional argument: the value wrapped by `l`. If the lock is already locked by a different task/thread, wait for it to become available. When this function returns, the `lock` has been released, so the caller should not attempt to `unlock` it.

Julia 1.11

Requires at least Julia 1.11.

source

```
lock(f::Function, lock)
```

Acquire the `lock`, execute `f` with the `lock` held, and release the `lock` when `f` returns. If the lock is already locked by a different task/thread, wait for it to become available.

When this function returns, the `lock` has been released, so the caller should not attempt to `unlock` it.

See also: `@lock`.

Julia 1.7

Using a `Channel` as the second argument requires Julia 1.7 or later.

source

```
lock(lock)
```

Acquire the `lock` when it becomes available. If the lock is already locked by a different task/thread, wait for it to become available.

Each `lock` must be matched by an `unlock`.

source

`Base.unlock` – Function.

```
unlock(lock)
```

Releases ownership of the `lock`.

If this is a recursive lock which has been acquired before, decrement an internal counter and return immediately.

source

`Base.trylock` – Function.

```
trylock(lock) -> Success (Boolean)
```

Acquire the lock if it is available, and return `true` if successful. If the lock is already locked by a different task/thread, return `false`.

Each successful `trylock` must be matched by an `unlock`.

Function `trylock` combined with `islocked` can be used for writing the test-and-test-and-set or exponential backoff algorithms *if it is supported by the `typeof(lock)`* (read its documentation).

source

`Base.islocked` – Function.

```
islocked(lock) -> Status (Boolean)
```

Check whether the `lock` is held by any task/thread. This function alone should not be used for synchronization. However, `islocked` combined with `trylock` can be used for writing the test-and-test-and-set or exponential backoff algorithms *if it is supported by the `typeof(lock)`* (read its documentation).

Extended help

For example, an exponential backoff can be implemented as follows if the `lock` implementation satisfied the properties documented below.

```
nspins = 0
while true
    while islocked(lock)
        GC.safepoint()
        nspins += 1
        nspins > LIMIT && error("timeout")
    end
    trylock(lock) && break
    backoff()
end
```

Implementation

A lock implementation is advised to define `islocked` with the following properties and note it in its docstring.

- `islocked(lock)` is data-race-free.
- If `islocked(lock)` returns `false`, an immediate invocation of `trylock(lock)` must succeed (returns `true`) if there is no interference from other tasks.

source

`Base.ReentrantLock` – Type.

```
ReentrantLock()
```

Creates a re-entrant lock for synchronizing Tasks. The same task can acquire the lock as many times as required (this is what the "Reentrant" part of the name means). Each `lock` must be matched with an `unlock`.

Calling `lock` will also inhibit running of finalizers on that thread until the corresponding `unlock`. Use of the standard lock pattern illustrated below should naturally be supported, but beware of inverting the try/lock order or missing the try block entirely (e.g. attempting to return with the lock still held):

This provides a acquire/release memory ordering on lock/unlock calls.

```
lock(l)
try
    <atomic work>
finally
    unlock(l)
end
```

If `!islocked(lck::ReentrantLock)` holds, `trylock(lck)` succeeds unless there are other tasks attempting to hold the lock "at the same time."

source

`Base.@lock` – Macro.

```
@lock l expr
```

Macro version of `lock(f, l::AbstractLock)` but with `expr` instead of `f` function. Expands to:

```
lock(l)
try
    expr
finally
    unlock(l)
end
```

This is similar to using `lock` with a do block, but avoids creating a closure and thus can improve the performance.

Compat

`@lock` was added in Julia 1.3, and exported in Julia 1.7.

source

`Base.Lockable` – Type.

```
Lockable(value, lock = ReentrantLock())
```

Creates a `Lockable` object that wraps `value` and associates it with the provided `lock`. This object supports `@lock`, `lock`, `trylock`, `unlock`. To access the value, index the lockable object while holding the lock.

Julia 1.11

Requires at least Julia 1.11.

Example

```
julia> locked_list = Base.Lockable(Int[]);

julia> @lock(locked_list, push!(locked_list[], 1)) # must hold the lock to access the value
1-element Vector{Int64}:
 1

julia> lock(summary, locked_list)
"1-element Vector{Int64}"
```

source

50.3 Channels

`Base.AbstractChannel` – Type.

```
AbstractChannel{T}
```

Representation of a channel passing objects of type T.

source

`Base.Channel` – Type.

```
Channel{T=Any}(size::Int=0)
```

Constructs a Channel with an internal buffer that can hold a maximum of size objects of type T. put! calls on a full channel block until an object is removed with take!.

Channel(0) constructs an unbuffered channel. put! blocks until a matching take! is called. And vice-versa.

Other constructors:

- Channel(): default constructor, equivalent to Channel{Any}(0)
- Channel(Inf): equivalent to Channel{Any}(typemax(Int))
- Channel(sz): equivalent to Channel{Any}(sz)

Julia 1.3

The default constructor Channel() and default size=0 were added in Julia 1.3.

source

Base.Channel – Method.

```
Channel{T=Any}(func::Function, size=0; taskref=nothing, spawn=false, threadpool=nothing)
```

Create a new task from func, bind it to a new channel of type T and size size, and schedule the task, all in a single call. The channel is automatically closed when the task terminates.

func must accept the bound channel as its only argument.

If you need a reference to the created task, pass a Ref{Task} object via the keyword argument taskref.

If spawn=true, the Task created for func may be scheduled on another thread in parallel, equivalent to creating a task via Threads.@spawn.

If spawn=true and the threadpool argument is not set, it defaults to :default.

If the threadpool argument is set (to :default or :interactive), this implies that spawn=true and the new Task is spawned to the specified threadpool.

Return a Channel.

Examples

```
julia> chnl = Channel() do ch
           foreach(i -> put!(ch, i), 1:4)
       end;

julia> typeof(chnl)
Channel{Any}

julia> for i in chnl
           @show i
       end;
i = 1
i = 2
i = 3
i = 4
```

Referencing the created task:

```
julia> taskref = Ref{Task}();

julia> chnl = Channel(taskref=taskref) do ch
           println(take!(ch))
       end;

julia> istaskdone(taskref[])
false

julia> put!(chnl, "Hello");
Hello

julia> istaskdone(taskref[])
true
```

Julia 1.3

The spawn= parameter was added in Julia 1.3. This constructor was added in Julia 1.3. In earlier versions of Julia, Channel used keyword arguments to set size and T, but those constructors are deprecated.

Julia 1.9

The threadpool= argument was added in Julia 1.9.

```
julia> chnl = Channel{Char}(1, spawn=true) do ch
           for c in "hello world"
               put!(ch, c)
           end
       end;

julia> String(collect(chnl))
"hello world"
```

source

Base.put! – Method.

```
put!(c::Channel, v)
```

Append an item v to the channel c. Blocks if the channel is full.

For unbuffered channels, blocks until a take! is performed by a different task.

Julia 1.1

v now gets converted to the channel's type with convert as put! is called.

source

Base.take! – Method.

```
take!(c::Channel)
```

Removes and returns a value from a `Channel` in order. Blocks until data is available. For unbuffered channels, blocks until a `put!` is performed by a different task.

Examples

Buffered channel

```
julia> c = Channel(1);

julia> put!(c, 1);

julia> take!(c)
1
```

Unbuffered channel

```
julia> c = Channel(0);

julia> task = Task(() -> put!(c, 1));

julia> schedule(task);

julia> take!(c)
1
```

source

`Base.isfull` – Method.

```
isfull(c::Channel)
```

Determines if a `Channel` is full, in the sense that calling `put!(c, some_value)` would have blocked. Returns immediately, does not block.

Note that it may frequently be the case that `put!` will not block after this returns `true`. Users must take precautions not to accidentally create live-lock bugs in their code by calling this method, as these are generally harder to debug than deadlocks. It is also possible that `put!` will block after this call returns `false`, if there are multiple producer tasks calling `put!` in parallel.

Examples

Buffered channel

```
julia> c = Channel(1); # capacity = 1

julia> isfull(c)
false

julia> put!(c, 1);

julia> isfull(c)
true
```

Unbuffered channel

```
julia> c = Channel(); # capacity = 0

julia> isfull(c) # unbuffered channel is always full
true
```

source

Base.isready – Method.

```
isready(c::Channel)
```

Determines whether a Channel has a value stored in it. Returns immediately, does not block.

For unbuffered channels, return `true` if there are tasks waiting on a `put!`.

Examples

Buffered channel

```
julia> c = Channel(1);

julia> isready(c)
false

julia> put!(c, 1);

julia> isready(c)
true
```

Unbuffered channel

```
julia> c = Channel();

julia> isready(c)  # no tasks waiting to put!
false

julia> task = Task(() -> put!(c, 1));

julia> schedule(task);  # schedule a put! task

julia> isready(c)
true
```

source

Base.isopen – Method.

```
isopen(c::Channel)
```

Determines whether a Channel is open for new `put!` operations. Notice that a Channel can be closed and still have buffered elements which can be consumed with `take!`.

Examples

Buffered channel with task

```
julia> c = Channel(ch -> put!(ch, 1), 1);

julia> isopen(c) # The channel is closed to new `put!`s
false

julia> isready(c) # The channel is closed but still contains elements
true

julia> take!(c)
1

julia> isready(c)
false
```

Unbuffered channel

```
julia> c = Channel{Int}();

julia> isopen(c)
true

julia> close(c)

julia> isopen(c)
false
```

source

`Base.fetch` – Method.

```
fetch(c::Channel)
```

Waits for and returns (without removing) the first available item from the `Channel`. Note: `fetch` is unsupported on an unbuffered (0-size) `Channel`.

Examples

Buffered channel

```
julia> c = Channel(3) do ch
           foreach(i -> put!(ch, i), 1:3)
       end;

julia> fetch(c)
1

julia> collect(c)  # item is not removed
3-element Vector{Any}:
 1
 2
 3
```

source

Base.close – Method.

```
close(c::Channel[, excp::Exception])
```

Close a channel. An exception (optionally given by excp), is thrown by:

- put! on a closed channel.
- take! and fetch on an empty, closed channel.

source

Base.bind – Method.

```
bind(chnl::Channel, task::Task)
```

Associate the lifetime of chnl with a task. Channel chnl is automatically closed when the task terminates. Any uncaught exception in the task is propagated to all waiters on chnl.

The chnl object can be explicitly closed independent of task termination. Terminating tasks have no effect on already closed Channel objects.

When a channel is bound to multiple tasks, the first task to terminate will close the channel. When multiple channels are bound to the same task, termination of the task will close all of the bound channels.

Examples

```
julia> c = Channel(0);

julia> task = @async foreach(i->put!(c, i), 1:4);

julia> bind(c,task);

julia> for i in c
           @show i
       end;
i = 1
i = 2
i = 3
i = 4

julia> isopen(c)
false
```

```
julia> c = Channel(0);

julia> task = @async (put!(c, 1); error("foo"));

julia> bind(c, task);

julia> take!(c)
1

julia> put!(c, 1);
```

```
ERROR: TaskFailedException
Stacktrace:
[...]
    nested task error: foo
[...]
```

source

50.4 Low-level synchronization using `schedule` and `wait`

The easiest correct use of `schedule` is on a `Task` that is not started (scheduled) yet. However, it is possible to use `schedule` and `wait` as a very low-level building block for constructing synchronization interfaces. A crucial pre-condition of calling `schedule(task)` is that the caller must "own" the `task`; i.e., it must know that the call to `wait` in the given `task` is happening at the locations known to the code calling `schedule(task)`. One strategy for ensuring such pre-condition is to use atomics, as demonstrated in the following example:

```
@enum OWEState begin
    OWE_EMPTY
    OWE_WAITING
    OWE_NOTIFYING
end

mutable struct OneWayEvent
    @atomic state::OWEState
    task::Task
    OneWayEvent() = new(OWE_EMPTY)
end

function Base.notify(ev::OneWayEvent)
    state = @atomic ev.state
    while state !== OWE_NOTIFYING
        # Spin until we successfully update the state to OWE_NOTIFYING:
        state, ok = @atomicreplace(ev.state, state => OWE_NOTIFYING)
        if ok
            if state == OWE_WAITING
                # OWE_WAITING -> OWE_NOTIFYING transition means that the waiter task is
                # already waiting or about to call `wait`. The notifier task must wake up
                # the waiter task.
                schedule(ev.task)
            else
                @assert state == OWE_EMPTY
                # Since we are assuming that there is only one notifier task (for
                # simplicity), we know that the other possible case here is OWE_EMPTY.
                # We do not need to do anything because we know that the waiter task has
                # not called `wait(ev::OneWayEvent)` yet.
            end
            break
        end
    end
    return
end
```

```
function Base.wait(ev::OneWayEvent)
    ev.task = current_task()
    state, ok = @atomicreplace(ev.state, OWE_EMPTY => OWE_WAITING)
    if ok
        # OWE_EMPTY -> OWE_WAITING transition means that the notifier task is guaranteed to
        # invoke OWE_WAITING -> OWE_NOTIFYING transition. The waiter task must call
        # `wait()` immediately. In particular, it MUST NOT invoke any function that may
        # yield to the scheduler at this point in code.
        wait()
    else
        @assert state == OWE_NOTIFYING
        # Otherwise, the `state` must have already been moved to OWE_NOTIFYING by the
        # notifier task.
    end
    return
end

ev = OneWayEvent()
@sync begin
    Threads.@spawn begin
        wait(ev)
        println("done")
    end
    println("notifying...")
    notify(ev)
end

# output
notifying...
done
```

`OneWayEvent` lets one task to `wait` for another task's `notify`. It is a limited communication interface since `wait` can only be used once from a single task (note the non-atomic assignment of `ev.task`)

In this example, `notify(ev::OneWayEvent)` is allowed to call `schedule(ev.task)` if and only if *it* modifies the state from `OWE_WAITING` to `OWE_NOTIFYING`. This lets us know that the task executing `wait(ev::OneWayEvent)` is now in the `ok` branch and that there cannot be other tasks that try to `schedule(ev.task)` since their `@atomicreplace(ev.state, state => OWE_NOTIFYING)` will fail.

Chapter 51

Multi-Threading

`Base.Threads.@threads` – Macro.

```
Threads.@threads [schedule] for ... end
```

A macro to execute a `for` loop in parallel. The iteration space is distributed to coarse-grained tasks. This policy can be specified by the `schedule` argument. The execution of the loop waits for the evaluation of all iterations.

Tasks spawned by `@threads` are scheduled on the `:default` threadpool. This means that `@threads` will not use threads from the `:interactive` threadpool, even if called from the main thread or from a task in the interactive pool. The `:default` threadpool is intended for compute-intensive parallel workloads.

See also: `@spawn` and `pmap` in `Distributed`. For more information on threadpools, see the chapter on threadpools.

Extended help

Semantics

Unless stronger guarantees are specified by the scheduling option, the loop executed by `@threads` macro have the following semantics.

The `@threads` macro executes the loop body in an unspecified order and potentially concurrently. It does not specify the exact assignments of the tasks and the worker threads. The assignments can be different for each execution. The loop body code (including any code transitively called from it) must not make any assumptions about the distribution of iterations to tasks or the worker thread in which they are executed. The loop body for each iteration must be able to make forward progress independent of other iterations and be free from data races. As such, invalid synchronizations across iterations may deadlock while unsynchronized memory accesses may result in undefined behavior.

For example, the above conditions imply that:

- A lock taken in an iteration *must* be released within the same iteration.
- Communicating between iterations using blocking primitives like `Channels` is incorrect.
- Write only to locations not shared across iterations (unless a lock or atomic operation is used).
- Unless the `:static` schedule is used, the value of `threadid()` may change even within a single iteration. See `Task Migration`.

Schedulers

Without the scheduler argument, the exact scheduling is unspecified and varies across Julia releases. Currently, `:dynamic` is used when the scheduler is not specified.

Julia 1.5

The `schedule` argument is available as of Julia 1.5.

`:dynamic` (default)

`:dynamic` scheduler executes iterations dynamically to available worker threads. Current implementation assumes that the workload for each iteration is uniform. However, this assumption may be removed in the future.

This scheduling option is merely a hint to the underlying execution mechanism. However, a few properties can be expected. The number of `Task`s used by `:dynamic` scheduler is bounded by a small constant multiple of the number of available worker threads (`Threads.threadpoolsize()`). Each task processes contiguous regions of the iteration space. Thus, `@threads :dynamic for x in xs; f(x); end` is typically more efficient than `@sync for x in xs; @spawn f(x); end` if `length(xs)` is significantly larger than the number of the worker threads and the run-time of `f(x)` is relatively smaller than the cost of spawning and synchronizing a task (typically less than 10 microseconds).

Julia 1.8

The `:dynamic` option for the `schedule` argument is available and the default as of Julia 1.8.

`:greedy`

`:greedy` scheduler spawns up to `Threads.threadpoolsize()` tasks, each greedily working on the given iterated values as they are produced. As soon as one task finishes its work, it takes the next value from the iterator. Work done by any individual task is not necessarily on contiguous values from the iterator. The given iterator may produce values forever, only the iterator interface is required (no indexing).

This scheduling option is generally a good choice if the workload of individual iterations is not uniform/has a large spread.

Julia 1.11

The `:greedy` option for the `schedule` argument is available as of Julia 1.11.

`:static`

`:static` scheduler creates one task per thread and divides the iterations equally among them, assigning each task specifically to each thread. In particular, the value of `threadid()` is guaranteed to be constant within one iteration. Specifying `:static` is an error if used from inside another `@threads` loop or from a thread other than 1.

Note

`:static` scheduling exists for supporting transition of code written before Julia 1.3. In newly written library functions, `:static` scheduling is discouraged because the functions using this option cannot be called from arbitrary worker threads.

Examples

To illustrate of the different scheduling strategies, consider the following function `busywait` containing a non-yielding timed loop that runs for a given number of seconds.

```
julia> function busywait(seconds)
            tstart = time_ns()
            while (time_ns() - tstart) / 1e9 < seconds
            end
        end

julia> @time begin
            Threads.@spawn busywait(5)
            Threads.@threads :static for i in 1:Threads.threadpoolsize()
                busywait(1)
            end
        end
6.003001 seconds (16.33 k allocations: 899.255 KiB, 0.25% compilation time)

julia> @time begin
            Threads.@spawn busywait(5)
            Threads.@threads :dynamic for i in 1:Threads.threadpoolsize()
                busywait(1)
            end
        end
2.012056 seconds (16.05 k allocations: 883.919 KiB, 0.66% compilation time)
```

The `:dynamic` example takes 2 seconds since one of the non-occupied threads is able to run two of the 1-second iterations to complete the for loop.

source

`Base.Threads.foreach` – Function.

```
Threads.foreach(f, channel::Channel;
                schedule::Threads.AbstractSchedule=Threads.FairSchedule(),
                ntasks=Threads.threadpoolsize())
```

Similar to `foreach(f, channel)`, but iteration over `channel` and calls to `f` are split across `ntasks` tasks spawned by `Threads.@spawn`. This function will wait for all internally spawned tasks to complete before returning.

If `schedule isa FairSchedule`, `Threads.foreach` will attempt to spawn tasks in a manner that enables Julia's scheduler to more freely load-balance work items across threads. This approach generally has higher per-item overhead, but may perform better than `StaticSchedule` in concurrence with other multithreaded workloads.

If `schedule isa StaticSchedule`, `Threads.foreach` will spawn tasks in a manner that incurs lower per-item overhead than `FairSchedule`, but is less amenable to load-balancing. This approach thus may be more suitable for fine-grained, uniform workloads, but may perform worse than `FairSchedule` in concurrence with other multithreaded workloads.

Examples

```
julia> n = 20

julia> c = Channel{Int}(ch -> foreach(i -> put!(ch, i), 1:n), 1)

julia> d = Channel{Int}(n) do ch
           f = i -> put!(ch, i^2)
           Threads.foreach(f, c)
       end

julia> collect(d)
collect(d) = [1, 4, 9, 16, 25, 36, 49, 64, 81, 100, 121, 144, 169, 196, 225, 256, 289, 324,
↪  361, 400]
```

Julia 1.6

This function requires Julia 1.6 or later.

source

Base.Threads.@spawn – Macro.

```
Threads.@spawn [:default|:interactive|:samepool] expr
```

Create a Task and schedule it to run on any available thread in the specified threadpool: `:default`, `:interactive`, or `:samepool` to use the same as the caller. `:default` is used if unspecified. The task is allocated to a thread once one becomes available. To wait for the task to finish, call wait on the result of this macro, or call fetch to wait and then obtain its return value.

Values can be interpolated into @spawn via $, which copies the value directly into the constructed underlying closure. This allows you to insert the *value* of a variable, isolating the asynchronous code from changes to the variable's value in the current task.

Note

The thread that the task runs on may change if the task yields, therefore `threadid()` should not be treated as constant for a task. See Task Migration, and the broader multi-threading manual for further important caveats. See also the chapter on threadpools.

Julia 1.3

This macro is available as of Julia 1.3.

Julia 1.4

Interpolating values via $ is available as of Julia 1.4.

Julia 1.9

A threadpool may be specified as of Julia 1.9.

Julia 1.12

The same threadpool may be specified as of Julia 1.12.

Examples

```
julia> t() = println("Hello from ", Threads.threadid());

julia> tasks = fetch.([Threads.@spawn t() for i in 1:4]);
Hello from 1
Hello from 1
Hello from 3
Hello from 4
```

source

Base.Threads.threadid – Function.

```
Threads.threadid([t::Task])::Int
```

Get the ID number of the current thread of execution, or the thread of task `t`. The master thread has ID 1.

Examples

```
julia> Threads.threadid()
1

julia> Threads.@threads for i in 1:4
          println(Threads.threadid())
       end
4
2
5
4

julia> Threads.threadid(Threads.@spawn "foo")
2
```

Note

The thread that a task runs on may change if the task yields, which is known as Task Migration. For this reason in most cases it is not safe to use `threadid([task])` to index into, say, a vector of buffers or stateful objects.

source

Base.Threads.maxthreadid – Function.

```
Threads.maxthreadid()::Int
```

Get a lower bound on the number of threads (across all thread pools) available to the Julia process, with atomic-acquire semantics. The result will always be greater than or equal to `threadid()` as well as `threadid(task)` for any task you were able to observe before calling `maxthreadid`.

source

Base.Threads.nthreads – Function.

```
Threads.nthreads(:default | :interactive)::Int
```

Get the current number of threads within the specified thread pool. The threads in :interactive have id numbers 1:nthreads(:interactive), and the threads in :default have id numbers in nthreads(:interactive) .+ (1:nthreads(:default)).

See also BLAS.get_num_threads and BLAS.set_num_threads in the LinearAlgebra standard library, and nprocs() in the Distributed standard library and Threads.maxthreadid().

source

Base.Threads.threadpool – Function.

```
Threads.threadpool(tid = threadid())::Symbol
```

Returns the specified thread's threadpool; either :default, :interactive, or :foreign.

source

Base.Threads.nthreadpools – Function.

```
Threads.nthreadpools()::Int
```

Returns the number of threadpools currently configured.

source

Base.Threads.threadpoolsize – Function.

```
Threads.threadpoolsize(pool::Symbol = :default)::Int
```

Get the number of threads available to the default thread pool (or to the specified thread pool).

See also: BLAS.get_num_threads and BLAS.set_num_threads in the LinearAlgebra standard library, and nprocs() in the Distributed standard library.

source

Base.Threads.ngcthreads – Function.

```
Threads.ngcthreads()::Int
```

Returns the number of GC threads currently configured. This includes both mark threads and concurrent sweep threads.

source

See also Multi-Threading.

51.1 Atomic operations

`atomic` – Keyword.

Unsafe pointer operations are compatible with loading and storing pointers declared with `_Atomic` and `std::atomic` type in C11 and C++23 respectively. An error may be thrown if there is not support for atomically loading the Julia type T.

See also: `unsafe_load`, `unsafe_modify!`, `unsafe_replace!`, `unsafe_store!`, `unsafe_swap!`

`source`

`Base.@atomic` – Macro.

```
@atomic var
@atomic order ex
```

Mark `var` or ex as being performed atomically, if ex is a supported expression. If no `order` is specified it defaults to :sequentially_consistent.

```
@atomic a.b.x = new
@atomic a.b.x += addend
@atomic :release a.b.x = new
@atomic :acquire_release a.b.x += addend
@atomic m[idx] = new
@atomic m[idx] += addend
@atomic :release m[idx] = new
@atomic :acquire_release m[idx] += addend
```

Perform the store operation expressed on the right atomically and return the new value.

With assignment (=), this operation translates to a `setproperty!(a.b, :x, new)` or, in case of reference, to a `setindex_atomic!(m, order, new, idx)` call, with `order` defaulting to `:sequentially_consistent`.

With any modifying operator this operation translates to a `modifyproperty!(a.b, :x, op, addend)[2]` or, in case of reference, to a `modifyindex_atomic!(m, order, op, addend, idx...)[2]` call, with `order` defaulting to `:sequentially_consistent`.

```
@atomic a.b.x max arg2
@atomic a.b.x + arg2
@atomic max(a.b.x, arg2)
@atomic :acquire_release max(a.b.x, arg2)
@atomic :acquire_release a.b.x + arg2
@atomic :acquire_release a.b.x max arg2
@atomic m[idx] max arg2
@atomic m[idx] + arg2
@atomic max(m[idx], arg2)
@atomic :acquire_release max(m[idx], arg2)
@atomic :acquire_release m[idx] + arg2
@atomic :acquire_release m[idx] max arg2
```

Perform the binary operation expressed on the right atomically. Store the result into the field or the reference in the first argument, and return the values `(old, new)`.

This operation translates to a `modifyproperty!(a.b, :x, func, arg2)` or, in case of reference to a `modifyindex_atomic!(m, order, func, arg2, idx)` call, with `order` defaulting to `:sequentially_consistent`.

See Per-field atomics section in the manual for more details.

Examples

```
julia> mutable struct Atomic{T}; @atomic x::T; end

julia> a = Atomic(1)
Atomic{Int64}(1)

julia> @atomic a.x # fetch field x of a, with sequential consistency
1

julia> @atomic :sequentially_consistent a.x = 2 # set field x of a, with sequential
↪  consistency
2

julia> @atomic a.x += 1 # increment field x of a, with sequential consistency
3

julia> @atomic a.x + 1 # increment field x of a, with sequential consistency
3 => 4

julia> @atomic a.x # fetch field x of a, with sequential consistency
4

julia> @atomic max(a.x, 10) # change field x of a to the max value, with sequential
↪  consistency
4 => 10

julia> @atomic a.x max 5 # again change field x of a to the max value, with sequential
↪  consistency
10 => 10
```

```
julia> mem = AtomicMemory{Int}(undef, 2);

julia> @atomic mem[1] = 2 # set mem[1] to value 2 with sequential consistency
2

julia> @atomic :monotonic mem[1] # fetch the first value of mem, with monotonic consistency
2

julia> @atomic mem[1] += 1 # increment the first value of mem, with sequential consistency
3

julia> @atomic mem[1] + 1 # increment the first value of mem, with sequential consistency
3 => 4

julia> @atomic mem[1] # fetch the first value of mem, with sequential consistency
4

julia> @atomic max(mem[1], 10) # change the first value of mem to the max value, with
↪  sequential consistency
```

```
4 => 10

julia> @atomic mem[1] max 5 # again change the first value of mem to the max value, with
↪  sequential consistency
10 => 10
```

Julia 1.7

Atomic fields functionality requires at least Julia 1.7.

Julia 1.12

Atomic reference functionality requires at least Julia 1.12.

source

`Base.@atomicswap` – Macro.

```
@atomicswap a.b.x = new
@atomicswap :sequentially_consistent a.b.x = new
@atomicswap m[idx] = new
@atomicswap :sequentially_consistent m[idx] = new
```

Stores new into `a.b.x` (`m[idx]` in case of reference) and returns the old value of `a.b.x` (the old value stored at `m[idx]`, respectively).

This operation translates to a `swapproperty!(a.b, :x, new)` or, in case of reference, `swapindex_atomic!(mem, order, new, idx)` call, with `order` defaulting to `:sequentially_consistent`.

See Per-field atomics section in the manual for more details.

Examples

```
julia> mutable struct Atomic{T}; @atomic x::T; end

julia> a = Atomic(1)
Atomic{Int64}(1)

julia> @atomicswap a.x = 2+2 # replace field x of a with 4, with sequential consistency
1

julia> @atomic a.x # fetch field x of a, with sequential consistency
4
```

```
julia> mem = AtomicMemory{Int}(undef, 2);

julia> @atomic mem[1] = 1;

julia> @atomicswap mem[1] = 4 # replace the first value of `mem` with 4, with sequential
↪  consistency
1

julia> @atomic mem[1] # fetch the first value of mem, with sequential consistency
4
```

Julia 1.7

Atomic fields functionality requires at least Julia 1.7.

Julia 1.12

Atomic reference functionality requires at least Julia 1.12.

source

`Base.@atomicreplace` – Macro.

```
@atomicreplace a.b.x expected => desired
@atomicreplace :sequentially_consistent a.b.x expected => desired
@atomicreplace :sequentially_consistent :monotonic a.b.x expected => desired
@atomicreplace m[idx] expected => desired
@atomicreplace :sequentially_consistent m[idx] expected => desired
@atomicreplace :sequentially_consistent :monotonic m[idx] expected => desired
```

Perform the conditional replacement expressed by the pair atomically, returning the values `(old, success::Bool)`. Where `success` indicates whether the replacement was completed.

This operation translates to a `replaceproperty!(a.b, :x, expected, desired)` or, in case of reference, to a `replaceindex_atomic!(mem, success_order, fail_order, expected, desired, idx)` call, with both orders defaulting to `:sequentially_consistent`.

See Per-field atomics section in the manual for more details.

Examples

```
julia> mutable struct Atomic{T}; @atomic x::T; end

julia> a = Atomic(1)
Atomic{Int64}(1)

julia> @atomicreplace a.x 1 => 2 # replace field x of a with 2 if it was 1, with sequential
↪  consistency
(old = 1, success = true)

julia> @atomic a.x # fetch field x of a, with sequential consistency
2

julia> @atomicreplace a.x 1 => 3 # replace field x of a with 2 if it was 1, with sequential
↪  consistency
(old = 2, success = false)

julia> xchg = 2 => 0; # replace field x of a with 0 if it was 2, with sequential consistency

julia> @atomicreplace a.x xchg
(old = 2, success = true)

julia> @atomic a.x # fetch field x of a, with sequential consistency
0
```

```
julia> mem = AtomicMemory{Int}(undef, 2);

julia> @atomic mem[1] = 1;

julia> @atomicreplace mem[1] 1 => 2 # replace the first value of mem with 2 if it was 1, with
↪  sequential consistency
(old = 1, success = true)

julia> @atomic mem[1] # fetch the first value of mem, with sequential consistency
2

julia> @atomicreplace mem[1] 1 => 3 # replace field x of a with 2 if it was 1, with sequential
↪  consistency
(old = 2, success = false)

julia> xchg = 2 => 0; # replace field x of a with 0 if it was 2, with sequential consistency

julia> @atomicreplace mem[1] xchg
(old = 2, success = true)

julia> @atomic mem[1] # fetch the first value of mem, with sequential consistency
0
```

Julia 1.7

Atomic fields functionality requires at least Julia 1.7.

Julia 1.12

Atomic reference functionality requires at least Julia 1.12.

source

`Base.@atomiconce` – Macro.

```
@atomiconce a.b.x = value
@atomiconce :sequentially_consistent a.b.x = value
@atomiconce :sequentially_consistent :monotonic a.b.x = value
@atomiconce m[idx] = value
@atomiconce :sequentially_consistent m[idx] = value
@atomiconce :sequentially_consistent :monotonic m[idx] = value
```

Perform the conditional assignment of value atomically if it was previously unset. Returned value `success::Bool` indicates whether the assignment was completed.

This operation translates to a `setpropertyonce!(a.b, :x, value)` or, in case of reference, to a `setindexonce_atomic!(m, success_order, fail_order, value, idx)` call, with both orders defaulting to `:sequentially_consistent`.

See Per-field atomics section in the manual for more details.

Examples

```
julia> mutable struct AtomicOnce
```

```
            @atomic x
            AtomicOnce() = new()
        end

julia> a = AtomicOnce()
AtomicOnce(#undef)

julia> @atomiconce a.x = 1 # set field x of a to 1, if unset, with sequential consistency
true

julia> @atomic a.x # fetch field x of a, with sequential consistency
1

julia> @atomiconce :monotonic a.x = 2 # set field x of a to 1, if unset, with monotonic
↪  consistence
false
```

```
julia> mem = AtomicMemory{Vector{Int}}(undef, 1);

julia> isassigned(mem, 1)
false

julia> @atomiconce mem[1] = [1] # set the first value of mem to [1], if unset, with sequential
↪  consistency
true

julia> isassigned(mem, 1)
true

julia> @atomic mem[1] # fetch the first value of mem, with sequential consistency
1-element Vector{Int64}:
 1

julia> @atomiconce :monotonic mem[1] = [2] # set the first value of mem to [2], if unset, with
↪  monotonic
false

julia> @atomic mem[1]
1-element Vector{Int64}:
 1
```

Julia 1.11

Atomic fields functionality requires at least Julia 1.11.

Julia 1.12

Atomic reference functionality requires at least Julia 1.12.

source

`Core.AtomicMemory` – Type.

```
AtomicMemory{T} == GenericMemory{:atomic, T, Core.CPU}
```

Fixed-size DenseVector{T}. Fetching of any of its individual elements is performed atomically (with :monotonic ordering by default).

Warning

The access to AtomicMemory must be done by either using the @atomic macro or the lower level interface functions: Base.getindex_atomic, Base.setindex_atomic!, Base.setindexonce_atomic!, Base.swapindex_atomic!, Base.modifyindex_atomic!, and Base.replaceindex_atomic!.

For details, see Atomic Operations as well as macros @atomic, @atomiconce, @atomicswap, and @atomicreplace.

Julia 1.11

This type requires Julia 1.11 or later.

Julia 1.12

Lower level interface functions or @atomic macro requires Julia 1.12 or later.

source

There are also optional memory ordering parameters for the unsafe set of functions, that select the C/C++-compatible versions of these atomic operations, if that parameter is specified to unsafe_load, unsafe_store!, unsafe_swap!, unsafe_replace!, and unsafe_modify!.

Warning

The following APIs are deprecated, though support for them is likely to remain for several releases.

Base.Threads.Atomic – Type.

```
Threads.Atomic{T}
```

Holds a reference to an object of type T, ensuring that it is only accessed atomically, i.e. in a thread-safe manner.

New atomic objects can be created from a non-atomic values; if none is specified, the atomic object is initialized with zero.

Atomic objects can be accessed using the [] notation:

Examples

```
julia> x = Threads.Atomic{Int}(3)
Base.Threads.Atomic{Int64}(3)

julia> x[] = 1
1

julia> x[]
1
```

Atomic operations use an `atomic_` prefix, such as `atomic_add!`, `atomic_xchg!`, etc.

source

`Base.Threads.atomic_cas!` – Function.

```
Threads.atomic_cas!(x::Atomic{T}, cmp::T, newval::T) where T
```

Atomically compare-and-set x

Atomically compares the value in x with `cmp`. If equal, write `newval` to x. Otherwise, leaves x unmodified. Returns the old value in x. By comparing the returned value to `cmp` (via ===) one knows whether x was modified and now holds the new value `newval`.

For further details, see LLVM's `cmpxchg` instruction.

This function can be used to implement transactional semantics. Before the transaction, one records the value in x. After the transaction, the new value is stored only if x has not been modified in the mean time.

Examples

```
julia> x = Threads.Atomic{Int}(3)
Base.Threads.Atomic{Int64}(3)

julia> Threads.atomic_cas!(x, 4, 2);

julia> x
Base.Threads.Atomic{Int64}(3)

julia> Threads.atomic_cas!(x, 3, 2);

julia> x
Base.Threads.Atomic{Int64}(2)
```

source

`Base.Threads.atomic_xchg!` – Function.

```
Threads.atomic_xchg!(x::Atomic{T}, newval::T) where T
```

Atomically exchange the value in x

Atomically exchanges the value in x with `newval`. Returns the **old** value.

For further details, see LLVM's `atomicrmw xchg` instruction.

Examples

```
julia> x = Threads.Atomic{Int}(3)
Base.Threads.Atomic{Int64}(3)

julia> Threads.atomic_xchg!(x, 2)
3

julia> x[]
2
```

source

Base.Threads.atomic_add! – Function.

```
Threads.atomic_add!(x::Atomic{T}, val::T) where T <: ArithmeticTypes
```

Atomically add `val` to `x`

Performs `x[] += val` atomically. Returns the **old** value. Not defined for `Atomic{Bool}`.

For further details, see LLVM's `atomicrmw add` instruction.

Examples

```
julia> x = Threads.Atomic{Int}(3)
Base.Threads.Atomic{Int64}(3)

julia> Threads.atomic_add!(x, 2)
3

julia> x[]
5
```

source

Base.Threads.atomic_sub! – Function.

```
Threads.atomic_sub!(x::Atomic{T}, val::T) where T <: ArithmeticTypes
```

Atomically subtract `val` from `x`

Performs `x[] -= val` atomically. Returns the **old** value. Not defined for `Atomic{Bool}`.

For further details, see LLVM's `atomicrmw sub` instruction.

Examples

```
julia> x = Threads.Atomic{Int}(3)
Base.Threads.Atomic{Int64}(3)

julia> Threads.atomic_sub!(x, 2)
3

julia> x[]
1
```

source

Base.Threads.atomic_and! – Function.

```
Threads.atomic_and!(x::Atomic{T}, val::T) where T
```

Atomically bitwise-and `x` with `val`

Performs `x[] &= val` atomically. Returns the **old** value.

For further details, see LLVM's `atomicrmw and` instruction.

Examples

```
julia> x = Threads.Atomic{Int}(3)
Base.Threads.Atomic{Int64}(3)

julia> Threads.atomic_and!(x, 2)
3

julia> x[]
2
```

source

Base.Threads.atomic_nand! – Function.

```
Threads.atomic_nand!(x::Atomic{T}, val::T) where T
```

Atomically bitwise-nand (not-and) x with val

Performs x[] = ~(x[] & val) atomically. Returns the **old** value.

For further details, see LLVM's atomicrmw nand instruction.

Examples

```
julia> x = Threads.Atomic{Int}(3)
Base.Threads.Atomic{Int64}(3)

julia> Threads.atomic_nand!(x, 2)
3

julia> x[]
-3
```

source

Base.Threads.atomic_or! – Function.

```
Threads.atomic_or!(x::Atomic{T}, val::T) where T
```

Atomically bitwise-or x with val

Performs x[] |= val atomically. Returns the **old** value.

For further details, see LLVM's atomicrmw or instruction.

Examples

```
julia> x = Threads.Atomic{Int}(5)
Base.Threads.Atomic{Int64}(5)

julia> Threads.atomic_or!(x, 7)
5

julia> x[]
7
```

source

Base.Threads.atomic_xor! – Function.

```
Threads.atomic_xor!(x::Atomic{T}, val::T) where T
```

Atomically bitwise-xor (exclusive-or) x with val

Performs x[] $= val atomically. Returns the **old** value.

For further details, see LLVM's atomicrmw xor instruction.

Examples

```
julia> x = Threads.Atomic{Int}(5)
Base.Threads.Atomic{Int64}(5)

julia> Threads.atomic_xor!(x, 7)
5

julia> x[]
2
```

source

Base.Threads.atomic_max! – Function.

```
Threads.atomic_max!(x::Atomic{T}, val::T) where T
```

Atomically store the maximum of x and val in x

Performs x[] = max(x[], val) atomically. Returns the **old** value.

For further details, see LLVM's atomicrmw max instruction.

Examples

```
julia> x = Threads.Atomic{Int}(5)
Base.Threads.Atomic{Int64}(5)

julia> Threads.atomic_max!(x, 7)
5

julia> x[]
7
```

source

Base.Threads.atomic_min! – Function.

```
Threads.atomic_min!(x::Atomic{T}, val::T) where T
```

Atomically store the minimum of x and val in x

Performs x[] = min(x[], val) atomically. Returns the **old** value.

For further details, see LLVM's atomicrmw min instruction.

Examples

```
julia> x = Threads.Atomic{Int}(7)
Base.Threads.Atomic{Int64}(7)

julia> Threads.atomic_min!(x, 5)
7

julia> x[]
5
```

source

Base.Threads.atomic_fence – Function.

```
Threads.atomic_fence()
```

Insert a sequential-consistency memory fence

Inserts a memory fence with sequentially-consistent ordering semantics. There are algorithms where this is needed, i.e. where an acquire/release ordering is insufficient.

This is likely a very expensive operation. Given that all other atomic operations in Julia already have acquire/release semantics, explicit fences should not be necessary in most cases.

For further details, see LLVM's `fence` instruction.

source

51.2 ccall using a libuv threadpool (Experimental)

Base.@threadcall – Macro.

```
@threadcall((cfunc, clib), rettype, (argtypes...), argvals...)
```

The `@threadcall` macro is called in the same way as `ccall` but does the work in a different thread. This is useful when you want to call a blocking C function without causing the current `julia` thread to become blocked. Concurrency is limited by size of the libuv thread pool, which defaults to 4 threads but can be increased by setting the `UV_THREADPOOL_SIZE` environment variable and restarting the `julia` process.

Note that the called function should never call back into Julia.

source

51.3 Low-level synchronization primitives

These building blocks are used to create the regular synchronization objects.

Base.Threads.AbstractSpinLock – Type.

```
abstract type AbstractSpinLock <: AbstractLock end
```

A non-reentrant, test-and-test-and-set spin lock. Recursive use will result in a deadlock. This kind of lock should only be used around code that takes little time to execute and does not block (e.g. perform I/O). In general, `ReentrantLock` should be used instead.

Each `lock` must be matched with an `unlock`. If `!islocked(lck::AbstractSpinLock)` holds, `trylock(lck)` succeeds unless there are other tasks attempting to hold the lock "at the same time."

Test-and-test-and-set spin locks are quickest up to about 30ish contending threads. If you have more contention than that, different synchronization approaches should be considered.

Julia 1.13

This type requires at least Julia 1.13.

source

`Base.Threads.SpinLock` – Type.

```
SpinLock() <: AbstractSpinLock
```

Spinlocks are not padded, and so may suffer from false sharing. See also `PaddedSpinLock`.

See the documentation for `AbstractSpinLock` regarding correct usage.

source

`Base.Threads.PaddedSpinLock` – Type.

```
PaddedSpinLock() <: AbstractSpinLock
```

PaddedSpinLocks are padded so that each is guaranteed to be on its own cache line, to avoid false sharing. See also `SpinLock`.

See the documentation for `AbstractSpinLock` regarding correct usage.

Julia 1.13

This type requires at least Julia 1.13.

source

51.4 Task metrics (Experimental)

`Base.Experimental.task_metrics` – Function.

```
Base.Experimental.task_metrics(::Bool)
```

Enable or disable the collection of per-task metrics. A `Task` created when `Base.Experimental.task_metrics(true)` is in effect will have `Base.Experimental.task_running_time_ns` and `Base.Experimental.task_wall_time_ns` timing information available.

Note

Task metrics can be enabled at start-up via the `--task-metrics=yes` command line option.

source

`Base.Experimental.task_running_time_ns` – Function.

```
Base.Experimental.task_running_time_ns(t::Task)::Union{UInt64, Nothing}
```

Return the total nanoseconds that the task `t` has spent running. This metric is only updated when `t` yields or completes unless `t` is the current task, in which it will be updated continuously. See also `Base.Experimental.task_wall_time_ns`.

Returns `nothing` if task timings are not enabled. See `Base.Experimental.task_metrics`.

This metric is from the Julia scheduler

A task may be running on an OS thread that is descheduled by the OS scheduler, this time still counts towards the metric.

Julia 1.12

This method was added in Julia 1.12.

source

`Base.Experimental.task_wall_time_ns` – Function.

```
Base.Experimental.task_wall_time_ns(t::Task)::Union{UInt64, Nothing}
```

Return the total nanoseconds that the task `t` was runnable. This is the time since the task first entered the run queue until the time at which it completed, or until the current time if the task has not yet completed. See also `Base.Experimental.task_running_time_ns`.

Returns `nothing` if task timings are not enabled. See `Base.Experimental.task_metrics`.

Julia 1.12

This method was added in Julia 1.12.

source

Chapter 52

Scoped Values

Scoped values provide an implementation of dynamic scoping in Julia.

Lexical scoping vs dynamic scoping

Lexical scoping is the default behavior in Julia. Under lexical scoping the scope of a variable is determined by the lexical (textual) structure of a program. Under dynamic scoping a variable is bound to the most recent assigned value during the program's execution.

The state of a scoped value is dependent on the execution path of the program. This means that for a scoped value you may observe multiple different values concurrently.

Julia 1.11

Scoped values were introduced in Julia 1.11. In Julia 1.8+ a compatible implementation is available from the package ScopedValues.jl.

In its simplest form you can create a `ScopedValue` with a default value and then use `with` or `@with` to enter a new dynamic scope. The new scope will inherit all values from the parent scope (and recursively from all outer scopes) with the provided scoped value taking priority over previous definitions.

Let's first look at an example of **lexical** scope. A `let` statement begins a new lexical scope within which the outer definition of x is shadowed by its inner definition.

```
julia> x = 1
1

julia> let x = 5
           @show x
       end;
x = 5

julia> @show x;
x = 1
```

In the following example, since Julia uses lexical scope, the variable x in the body of `f` refers to the x defined in the global scope, and entering a `let` scope does not change the value `f` observes.

```
julia> x = 1
1

julia> f() = @show x
f (generic function with 1 method)

julia> let x = 5
           f()
       end;
x = 1

julia> f();
x = 1
```

Now using a `ScopedValue` we can use **dynamic** scoping.

```
julia> using Base.ScopedValues

julia> x = ScopedValue(1)
ScopedValue{Int64}(1)

julia> f() = @show x[]
f (generic function with 1 method)

julia> with(x=>5) do
           f()
       end;
x[] = 5

julia> f();
x[] = 1
```

Note that the observed value of the `ScopedValue` is dependent on the execution path of the program.

It often makes sense to use a `const` variable to point to a scoped value, and you can set the value of multiple `ScopedValues` with one call to `with`.

```
using Base.ScopedValues

f() = @show a[]
g() = @show b[]

const a = ScopedValue(1)
const b = ScopedValue(2)

f() # a[] = 1
g() # b[] = 2

# Enter a new dynamic scope and set value.
with(a => 3) do
    f() # a[] = 3
    g() # b[] = 2
```

```
    with(a => 4, b => 5) do
        f() # a[] = 4
        g() # b[] = 5
    end
    f() # a[] = 3
    g() # b[] = 2
end

f() # a[] = 1
g() # b[] = 2
```

ScopedValues provides a macro version of `with`. The expression `@with var=>val expr` evaluates `expr` in a new dynamic scope with `var` set to `val`. `@with var=>val expr` is equivalent to `with(var=>val) do expr end`. However, `with` requires a zero-argument closure or function, which results in an extra call-frame. As an example, consider the following function `f`:

```
using Base.ScopedValues
const a = ScopedValue(1)
f(x) = a[] + x
```

If you wish to run `f` in a dynamic scope with a set to 2, then you can use `with`:

```
with(() -> f(10), a=>2)
```

However, this requires wrapping `f` in a zero-argument function. If you wish to avoid the extra call-frame, then you can use the `@with` macro:

```
@with a=>2 f(10)
```

Note

Dynamic scopes are inherited by Tasks, at the moment of task creation. Dynamic scopes are **not** propagated through `Distributed.jl` operations.

In the example below we open a new dynamic scope before launching a task. The parent task and the two child tasks observe independent values of the same scoped value at the same time.

```
using Base.ScopedValues
import Base.Threads: @spawn

const scoped_val = ScopedValue(1)
@sync begin
    with(scoped_val => 2)
        @spawn @show scoped_val[] # 2
    end
    with(scoped_val => 3)
        @spawn @show scoped_val[] # 3
    end
    @show scoped_val[] # 1
end
```

Scoped values are constant throughout a scope, but you can store mutable state in a scoped value. Just keep in mind that the usual caveats for global variables apply in the context of concurrent programming.

Care is also required when storing references to mutable state in scoped values. You might want to explicitly unshare mutable state when entering a new dynamic scope.

```
using Base.ScopedValues
import Base.Threads: @spawn

const sval_dict = ScopedValue(Dict())

# Example of using a mutable value wrongly
@sync begin
    # `Dict` is not thread-safe the usage below is invalid
    @spawn (sval_dict[][:a] = 3)
    @spawn (sval_dict[][:b] = 3)
end

@sync begin
    # If we instead pass a unique dictionary to each
    # task we can access the dictionaries race free.
    with(sval_dict => Dict()) do
        @spawn (sval_dict[][:a] = 3)
    end
    with(sval_dict => Dict()) do
        @spawn (sval_dict[][:b] = 3)
    end
end
```

52.1 Example

In the example below we use a scoped value to implement a permission check in a web-application. After determining the permissions of the request, a new dynamic scope is entered and the scoped value LEVEL is set. Other parts of the application can query the scoped value and will receive the appropriate value. Other alternatives like task-local storage and global variables are not well suited for this kind of propagation; our only alternative would have been to thread a value through the entire call-chain.

```
using Base.ScopedValues

const LEVEL = ScopedValue(:GUEST)

function serve(request, response)
    level = isAdmin(request) ? :ADMIN : :GUEST
    with(LEVEL => level) do
        Threads.@spawn handle(request, response)
    end
end

function open(connection::Database)
    level = LEVEL[]
    if level !== :ADMIN
        error("Access disallowed")
```

```
    end
    # ... open connection
end

function handle(request, response)
    # ...
    open(Database(#=...=#))
    # ...
end
```

52.2 Idioms

Unshare mutable state

```
using Base.ScopedValues
import Base.Threads: @spawn

const sval_dict = ScopedValue(Dict())

# If you want to add new values to the dict, instead of replacing
# it, unshare the values explicitly. In this example we use `merge`
# to unshare the state of the dictionary in parent scope.
@sync begin
    with(sval_dict => merge(sval_dict[], Dict(:a => 10))) do
        @spawn @show sval_dict[][:a]
    end
    @spawn sval_dict[][:a] = 3 # Not a race since they are unshared.
end
```

Scoped values as globals

In order to access the value of a scoped value, the scoped value itself has to be in (lexical) scope. This means most often you likely want to use scoped values as constant globals.

```
using Base.ScopedValues
const sval = ScopedValue(1)
```

Indeed one can think of scoped values as hidden function arguments.

This does not preclude their use as non-globals.

```
using Base.ScopedValues
import Base.Threads: @spawn

function main()
    role = ScopedValue(:client)

    function launch()
        #...
        role[]
    end
```

```
    @with role => :server @spawn launch()
    launch()
end
```

But it might have been simpler to just directly pass the function argument in these cases.

Very many ScopedValues

If you find yourself creating many `ScopedValue`'s for one given module, it may be better to use a dedicated struct to hold them.

```
using Base.ScopedValues

Base.@kwdef struct Configuration
    color::Bool = false
    verbose::Bool = false
end

const CONFIG = ScopedValue(Configuration(color=true))

@with CONFIG => Configuration(color=CONFIG[].color, verbose=true) begin
    @show CONFIG[].color # true
    @show CONFIG[].verbose # true
end
```

52.3 API docs

`Base.ScopedValues.AbstractScopedValue` – Type.

```
AbstractScopedValue{T}
```

Abstract base type for scoped values that propagate values across dynamic scopes. All scoped value types must extend this abstract type.

See also: `ScopedValue`, `LazyScopedValue`

Julia 1.13

AbstractScopedValue requires Julia 1.13+.

source

`Base.ScopedValues.ScopedValue` – Type.

```
ScopedValue(x)
```

Create a container that propagates values across dynamic scopes. Use `with` to create and enter a new dynamic scope.

Values can only be set when entering a new dynamic scope, and the value referred to will be constant during the execution of a dynamic scope.

Dynamic scopes are propagated across tasks.

Examples

```
julia> using Base.ScopedValues;

julia> const sval = ScopedValue(1);

julia> sval[]
1

julia> with(sval => 2) do
           sval[]
       end
2

julia> sval[]
1
```

Julia 1.11

Scoped values were introduced in Julia 1.11. In Julia 1.8+ a compatible implementation is available from the package ScopedValues.jl.

source

`Base.ScopedValues.LazyScopedValue` – Type.

```
LazyScopedValue{T}(f::OncePerProcess{T})
```

A scoped value that uses an `OncePerProcess{T}` to lazily compute its default value when none has been set in the current scope. Unlike `ScopedValue`, the default is not evaluated at construction time but only when first accessed.

Examples

```
julia> using Base.ScopedValues;

julia> const editor = LazyScopedValue(OncePerProcess(() -> ENV["JULIA_EDITOR"]));

julia> editor[]
"vim"

julia> with(editor => "emacs") do
           sval[]
       end
"emacs"

julia> editor[]
"vim"
```

Julia 1.13

LazyScopedValue requires Julia 1.13+.

source

Base.ScopedValues.with – Function.

```
with(f, (var::ScopedValue{T} => val)...)
```

Execute `f` in a new dynamic scope with `var` set to `val`. `val` will be converted to type `T`.

See also: `ScopedValues.@with`, `ScopedValues.ScopedValue`, `ScopedValues.get`.

Examples

```
julia> using Base.ScopedValues

julia> a = ScopedValue(1);

julia> f(x) = a[] + x;

julia> f(10)
11

julia> with(a=>2) do
           f(10)
       end
12

julia> f(10)
11

julia> b = ScopedValue(2);

julia> g(x) = a[] + b[] + x;

julia> with(a=>10, b=>20) do
           g(30)
       end
60

julia> with(() -> a[] * b[], a=>3, b=>4)
12
```

source

Base.ScopedValues.@with – Macro.

```
@with (var::ScopedValue{T} => val)... expr
```

Macro version of `with`. The expression `@with var=>val expr` evaluates `expr` in a new dynamic scope with `var` set to `val`. `val` will be converted to type `T`. `@with var=>val expr` is equivalent to `with(var=>val) do expr end`, but `@with` avoids creating a closure.

See also: `ScopedValues.with`, `ScopedValues.ScopedValue`, `ScopedValues.get`.

Examples

```
julia> using Base.ScopedValues
```

```
julia> const a = ScopedValue(1);

julia> f(x) = a[] + x;

julia> @with a=>2 f(10)
12

julia> @with a=>3 begin
           x = 100
           f(x)
       end
103
```

source

Base.isassigned – Method.

```
isassigned(val::ScopedValue)
```

Test whether a ScopedValue has an assigned value.

See also: ScopedValues.with, ScopedValues.@with, ScopedValues.get.

Examples

```
julia> using Base.ScopedValues

julia> a = ScopedValue(1); b = ScopedValue{Int}();

julia> isassigned(a)
true

julia> isassigned(b)
false
```

source

Base.ScopedValues.get – Function.

```
get(val::ScopedValue{T})::Union{Nothing, Some{T}}
get(val::LazyScopedValue{T})::Union{Nothing, Some{T}}
```

If the scoped value isn't set and doesn't have a default value, return nothing. Otherwise returns Some{T} with the current value.

See also: ScopedValues.with, ScopedValues.@with, ScopedValues.ScopedValue.

Examples

```
julia> using Base.ScopedValues

julia> a = ScopedValue(42); b = ScopedValue{Int}();

julia> ScopedValues.get(a)
Some(42)
```

```
julia> isnothing(ScopedValues.get(b))
true
```

source

52.4 Implementation notes and performance

Scopes use a persistent dictionary. Lookup and insertion is `O(log(32, n))`, upon dynamic scope entry a small amount of data is copied and the unchanged data is shared among other scopes.

The Scope object itself is not user-facing and may be changed in a future version of Julia.

52.5 Design inspiration

This design was heavily inspired by JEPS-429, which in turn was inspired by dynamically scoped free variables in many Lisp dialects. In particular Interlisp-D and its deep binding strategy.

A prior design discussed was context variables ala PEPS-567 and implemented in Julia as ContextVariablesX.jl.

Chapter 53

Constants

`Core.nothing` – Constant.

```
nothing
```

The singleton instance of type `Nothing`, used by convention when there is no value to return (as in a C `void` function) or when a variable or field holds no value.

A return value of `nothing` is not displayed by the REPL and similar interactive environments.

See also: `isnothing`, `something`, `missing`.

source

`Base.PROGRAM_FILE` – Constant.

```
PROGRAM_FILE
```

A string containing the script name passed to Julia from the command line. Note that the script name remains unchanged from within included files. Alternatively see `@__FILE__`.

source

`Base.ARGS` – Constant.

```
ARGS
```

An array of the command line arguments passed to Julia, as strings.

source

`Base.C_NULL` – Constant.

```
C_NULL
```

The C null pointer constant, sometimes used when calling external code.

source

`Base.VERSION` – Constant.

```
VERSION
```

A VersionNumber object describing which version of Julia is in use. See also Version Number Literals.

source

Base.DEPOT_PATH – Constant.

```
DEPOT_PATH
```

A stack of "depot" locations where the package manager, as well as Julia's code loading mechanisms, look for package registries, installed packages, named environments, repo clones, cached compiled package images, and configuration files. By default it includes:

1. ~/.julia where ~ is the user home as appropriate on the system;
2. an architecture-specific shared system directory, e.g. /usr/local/share/julia;
3. an architecture-independent shared system directory, e.g. /usr/share/julia.

So DEPOT_PATH might be:

```
[joinpath(homedir(), ".julia"), "/usr/local/share/julia", "/usr/share/julia"]
```

The first entry is the "user depot" and should be writable by and owned by the current user. The user depot is where: registries are cloned, new package versions are installed, named environments are created and updated, package repos are cloned, newly compiled package image files are saved, log files are written, development packages are checked out by default, and global configuration data is saved. Later entries in the depot path are treated as read-only and are appropriate for registries, packages, etc. installed and managed by system administrators.

DEPOT_PATH is populated based on the JULIA_DEPOT_PATH environment variable if set.

DEPOT_PATH contents

Each entry in DEPOT_PATH is a path to a directory which contains subdirectories used by Julia for various purposes. Here is an overview of some of the subdirectories that may exist in a depot:

- artifacts: Contains content that packages use for which Pkg manages the installation of.
- clones: Contains full clones of package repos. Maintained by Pkg.jl and used as a cache.
- config: Contains julia-level configuration such as a startup.jl.
- compiled: Contains precompiled *.ji files for packages. Maintained by Julia.
- dev: Default directory for Pkg.develop. Maintained by Pkg.jl and the user.
- environments: Default package environments. For instance the global environment for a specific julia version. Maintained by Pkg.jl.
- logs: Contains logs of Pkg and REPL operations. Maintained by Pkg.jl and Julia.
- packages: Contains packages, some of which were explicitly installed and some which are implicit dependencies. Maintained by Pkg.jl.
- registries: Contains package registries. By default only General. Maintained by Pkg.jl.
- scratchspaces: Contains content that a package itself installs via the Scratch.jl package. Pkg.gc() will delete content that is known to be unused.

Note

Packages that want to store content should use the `scratchspaces` subdirectory via `Scratch.jl` instead of creating new subdirectories in the depot root.

See also `JULIA_DEPOT_PATH`, and Code Loading.

source

`Base.LOAD_PATH` – Constant.

```
LOAD_PATH
```

An array of paths for `using` and `import` statements to consider as project environments or package directories when loading code. It is populated based on the `JULIA_LOAD_PATH` environment variable if set; otherwise it defaults to `["@", "@v#.#", "@stdlib"]`. Entries starting with `@` have special meanings:

- `@` refers to the "current active environment", the initial value of which is initially determined by the `JULIA_PROJECT` environment variable or the `--project` command-line option.
- `@stdlib` expands to the absolute path of the current Julia installation's standard library directory.
- `@name` refers to a named environment, which are stored in depots (see `JULIA_DEPOT_PATH`) under the `environments` subdirectory. The user's named environments are stored in `~/.julia/environments` so `@name` would refer to the environment in `~/.julia/environments/name` if it exists and contains a `Project.toml` file. If `name` contains `#` characters, then they are replaced with the major, minor and patch components of the Julia version number. For example, if you are running Julia 1.2 then `@v#.#` expands to `@v1.2` and will look for an environment by that name, typically at `~/.julia/environments/v1.2`.

The fully expanded value of `LOAD_PATH` that is searched for projects and packages can be seen by calling the `Base.load_path()` function.

See also `JULIA_LOAD_PATH`, `JULIA_PROJECT`, `JULIA_DEPOT_PATH`, and Code Loading.

source

`Base.Sys.BINDIR` – Constant.

```
Sys.BINDIR::String
```

A string containing the full path to the directory containing the `julia` executable.

source

`Base.Sys.CPU_THREADS` – Constant.

```
Sys.CPU_THREADS::Int
```

The number of logical CPU cores available in the system, i.e. the number of threads that the CPU can run concurrently. Note that this is not necessarily the number of CPU cores, for example, in the presence of hyper-threading.

See Hwloc.jl or CpuId.jl for extended information, including number of physical cores.

See also: `Sys.EFFECTIVE_CPU_THREADS` for a container-aware CPU count that respects cgroup limits.

source

Base.Sys.EFFECTIVE_CPU_THREADS – Constant.

```
Sys.EFFECTIVE_CPU_THREADS::Int
```

The effective number of logical CPU cores available to the Julia process, taking into account container limits (e.g., Docker `--cpus`, Kubernetes CPU limits, cgroup quotas). This is the minimum of the hardware CPU thread count and any imposed CPU limits.

In non-containerized environments, this typically equals `Sys.CPU_THREADS`. In containerized environments, it respects cgroup CPU limits and provides a more accurate measure of available parallelism.

Use this constant when determining default thread pool sizes or parallelism levels to ensure proper behavior in containerized deployments.

source

Base.Sys.WORD_SIZE – Constant.

```
Sys.WORD_SIZE::Int
```

Standard word size on the current machine, in bits.

source

Base.Sys.KERNEL – Constant.

```
Sys.KERNEL::Symbol
```

A symbol representing the name of the operating system, as returned by `uname` of the build configuration.

source

Base.Sys.ARCH – Constant.

```
Sys.ARCH::Symbol
```

A symbol representing the architecture of the build configuration.

source

Base.Sys.MACHINE – Constant.

```
Sys.MACHINE::String
```

A string containing the build triple.

source

See also:

- stdin
- stdout
- stderr
- ENV
- ENDIAN_BOM

Chapter 54

Filesystem

Base.read – Method.

```
read(filename::AbstractString)
```

Read the entire contents of a file as a `Vector{UInt8}`.

```
read(filename::AbstractString, String)
```

Read the entire contents of a file as a string.

```
read(filename::AbstractString, args...)
```

Open a file and read its contents. `args` is passed to `read`: this is equivalent to `open(io->read(io, args...), filename)`.

source

Base.write – Method.

```
write(filename::AbstractString, content)
```

Write the canonical binary representation of `content` to a file, which will be created if it does not exist yet or overwritten if it does exist.

Return the number of bytes written into the file.

source

Base.Filesystem.pwd – Function.

```
pwd()::String
```

Get the current working directory.

See also: cd, tempdir.

Examples

```
julia> pwd()
"/home/JuliaUser"

julia> cd("/home/JuliaUser/Projects/julia")

julia> pwd()
"/home/JuliaUser/Projects/julia"
```

source

Base.Filesystem.cd – Method.

```
cd(dir::AbstractString=homedir())
```

Set the current working directory.

See also: pwd, mkdir, mkpath, mktempdir.

Examples

```
julia> cd("/home/JuliaUser/Projects/julia")

julia> pwd()
"/home/JuliaUser/Projects/julia"

julia> cd()

julia> pwd()
"/home/JuliaUser"
```

source

Base.Filesystem.cd – Method.

```
cd(f::Function, dir::AbstractString=homedir())
```

Temporarily change the current working directory to `dir`, apply function `f` and finally return to the original directory.

Examples

```
julia> pwd()
"/home/JuliaUser"

julia> cd(readdir, "/home/JuliaUser/Projects/julia")
34-element Vector{String}:
 ".circleci"
 ".freebsdci.sh"
 ".git"
 ".gitattributes"
 ".github"
 □
 "test"
 "ui"
 "usr"
```

```
 "usr-staging"

julia> pwd()
"/home/JuliaUser"
```

source

Base.Filesystem.readdir – Function.

```
readdir(dir::AbstractString=pwd();
    join::Bool = false,
    sort::Bool = true,
)::Vector{String}
```

Return the names in the directory `dir` or the current working directory if not given. When `join` is false, `readdir` returns just the names in the directory as is; when `join` is true, it returns `joinpath(dir, name)` for each `name` so that the returned strings are full paths. If you want to get absolute paths back, call `readdir` with an absolute directory path and `join` set to true.

By default, `readdir` sorts the list of names it returns. If you want to skip sorting the names and get them in the order that the file system lists them, you can use `readdir(dir, sort=false)` to opt out of sorting.

See also: `walkdir`.

Julia 1.4

The `join` and `sort` keyword arguments require at least Julia 1.4.

Examples

```
julia> cd("/home/JuliaUser/dev/julia")

julia> readdir()
30-element Vector{String}:
 ".appveyor.yml"
 ".git"
 ".gitattributes"
 
 "ui"
 "usr"
 "usr-staging"

julia> readdir(join=true)
30-element Vector{String}:
 "/home/JuliaUser/dev/julia/.appveyor.yml"
 "/home/JuliaUser/dev/julia/.git"
 "/home/JuliaUser/dev/julia/.gitattributes"
 
 "/home/JuliaUser/dev/julia/ui"
 "/home/JuliaUser/dev/julia/usr"
 "/home/JuliaUser/dev/julia/usr-staging"

julia> readdir("base")
```

```
145-element Vector{String}:
 ".gitignore"
 "Base.jl"
 "Enums.jl"
 ⋮
 "version_git.sh"
 "views.jl"
 "weakkeydict.jl"

julia> readdir("base", join=true)
145-element Vector{String}:
 "base/.gitignore"
 "base/Base.jl"
 "base/Enums.jl"
 ⋮
 "base/version_git.sh"
 "base/views.jl"
 "base/weakkeydict.jl"

julia> readdir(abspath("base"), join=true)
145-element Vector{String}:
 "/home/JuliaUser/dev/julia/base/.gitignore"
 "/home/JuliaUser/dev/julia/base/Base.jl"
 "/home/JuliaUser/dev/julia/base/Enums.jl"
 ⋮
 "/home/JuliaUser/dev/julia/base/version_git.sh"
 "/home/JuliaUser/dev/julia/base/views.jl"
 "/home/JuliaUser/dev/julia/base/weakkeydict.jl"
```

source

`Base.Filesystem.walkdir` – Function.

```
walkdir(dir = pwd(); topdown=true, follow_symlinks=false, onerror=throw)
```

Return an iterator that walks the directory tree of a directory.

The iterator returns a tuple containing `(path, dirs, files)`. Each iteration `path` will change to the next directory in the tree; then `dirs` and `files` will be vectors containing the directories and files in the current `path` directory. The directory tree can be traversed top-down or bottom-up. If `walkdir` or `stat` encounters a `IOError` it will rethrow the error by default. A custom error handling function can be provided through `onerror` keyword argument. `onerror` is called with a `IOError` as argument. The returned iterator is stateful so when accessed repeatedly each access will resume where the last left off, like `Iterators.Stateful`.

See also: `readdir`.

Julia 1.12

`pwd()` as the default directory was added in Julia 1.12.

Examples

```
for (path, dirs, files) in walkdir(".")
```

```
    println("Directories in $path")
    for dir in dirs
        println(joinpath(path, dir)) # path to directories
    end
    println("Files in $path")
    for file in files
        println(joinpath(path, file)) # path to files
    end
end
```

```
julia> mkpath("my/test/dir");

julia> itr = walkdir("my");

julia> (path, dirs, files) = first(itr)
("my", ["test"], String[])

julia> (path, dirs, files) = first(itr)
("my/test", ["dir"], String[])

julia> (path, dirs, files) = first(itr)
("my/test/dir", String[], String[])
```

source

Base.Filesystem.mkdir – Function.

```
mkdir(path::AbstractString; mode::Unsigned = 0o777)
```

Make a new directory with name path and permissions mode. mode defaults to 0o777, modified by the current file creation mask. This function never creates more than one directory. If the directory already exists, or some intermediate directories do not exist, this function throws an error. See mkpath for a function which creates all required intermediate directories. Return path.

Examples

```
julia> mkdir("testingdir")
"testingdir"

julia> cd("testingdir")

julia> pwd()
"/home/JuliaUser/testingdir"
```

source

Base.Filesystem.mkpath – Function.

```
mkpath(path::AbstractString; mode::Unsigned = 0o777)
```

Create all intermediate directories in the path as required. Directories are created with the permissions mode which defaults to 0o777 and is modified by the current file creation mask. Unlike mkdir, mkpath does not error if path (or parts of it) already exists. However, an error will be thrown if path (or parts of it) points to an existing file. Return path.

If path includes a filename you will probably want to use `mkpath(dirname(path))` to avoid creating a directory using the filename.

Examples

```
julia> cd(mktempdir())

julia> mkpath("my/test/dir") # creates three directories
"my/test/dir"

julia> readdir()
1-element Vector{String}:
 "my"

julia> cd("my")

julia> readdir()
1-element Vector{String}:
 "test"

julia> readdir("test")
1-element Vector{String}:
 "dir"

julia> mkpath("intermediate_dir/actually_a_directory.txt") # creates two directories
"intermediate_dir/actually_a_directory.txt"

julia> isdir("intermediate_dir/actually_a_directory.txt")
true

julia> mkpath("my/test/dir/") # returns the original `path`
"my/test/dir/"
```

source

Base.Filesystem.hardlink – Function.

```
hardlink(src::AbstractString, dst::AbstractString)
```

Creates a hard link to an existing source file `src` with the name `dst`. The destination, `dst`, must not exist.

See also: `symlink`.

Julia 1.8

This method was added in Julia 1.8.

source

Base.Filesystem.symlink – Function.

```
symlink(target::AbstractString, link::AbstractString; dir_target = false)
```

Creates a symbolic link to `target` with the name `link`.

On Windows, symlinks must be explicitly declared as referring to a directory or not. If `target` already exists, by default the type of `link` will be auto- detected, however if `target` does not exist, this function defaults to creating a file symlink unless `dir_target` is set to `true`. Note that if the user sets `dir_target` but `target` exists and is a file, a directory symlink will still be created, but dereferencing the symlink will fail, just as if the user creates a file symlink (by calling `symlink()` with `dir_target` set to `false` before the directory is created) and tries to dereference it to a directory.

Additionally, there are two methods of making a link on Windows; symbolic links and junction points. Junction points are slightly more efficient, but do not support relative paths, so if a relative directory symlink is requested (as denoted by `isabspath(target)` returning `false`) a symlink will be used, else a junction point will be used. Best practice for creating symlinks on Windows is to create them only after the files/directories they reference are already created.

See also: `hardlink`.

Note

This function raises an error under operating systems that do not support soft symbolic links, such as Windows XP.

Julia 1.6

The `dir_target` keyword argument was added in Julia 1.6. Prior to this, symlinks to nonexistent paths on windows would always be file symlinks, and relative symlinks to directories were not supported.

source

`Base.Filesystem.readlink` – Function.

```
readlink(path::AbstractString)::String
```

Return the target location a symbolic link `path` points to.

source

`Base.Filesystem.chmod` – Function.

```
chmod(path::AbstractString, mode::Integer; recursive::Bool=false)
```

Change the permissions mode of `path` to `mode`. Only integer `mode`s (e.g. `0o777`) are currently supported. If `recursive=true` and the path is a directory all permissions in that directory will be recursively changed. Return `path`.

Note

Prior to Julia 1.6, this did not correctly manipulate filesystem ACLs on Windows, therefore it would only set read-only bits on files. It now is able to manipulate ACLs.

source

`Base.Filesystem.chown` – Function.

```
chown(path::AbstractString, owner::Integer, group::Integer=-1)
```

Change the owner and/or group of `path` to `owner` and/or `group`. If the value entered for `owner` or `group` is `-1` the corresponding ID will not change. Only integer `owners` and `groups` are currently supported. Return `path`.

source

`Base.Libc.RawFD` – Type.

```
RawFD
```

Primitive type which wraps the native OS file descriptor. RawFDs can be passed to methods like `stat` to discover information about the underlying file, and can also be used to open streams, with the RawFD describing the OS file backing the stream.

source

`Base.stat` – Function.

```
stat(path)
stat(path_elements...)
```

Return a structure whose fields contain information about the file. If multiple arguments are given, they are joined by `joinpath`.

The fields of the structure are:

Name	Type	Description
desc	`Union{String, Base.OS_HANDLE}`	The path or OS file descriptor
size	`Int64`	The size (in bytes) of the file
device	`UInt`	ID of the device that contains the file
inode	`UInt`	The inode number of the file
mode	`UInt`	The protection mode of the file
nlink	`Int`	The number of hard links to the file
uid	`UInt`	The user id of the owner of the file
gid	`UInt`	The group id of the file owner
rdev	`UInt`	If this file refers to a device, the ID of the device it refers to
blk-size	`Int64`	The file-system preferred block size for the file
blocks	`Int64`	The number of 512-byte blocks allocated
mtime	`Float64`	Unix timestamp of when the file was last modified
ctime	`Float64`	Unix timestamp of when the file's metadata was changed

source

`Base.Filesystem.diskstat` – Function.

```
diskstat(path=pwd())
```

Returns statistics in bytes about the disk that contains the file or directory pointed at by `path`. If no argument is passed, statistics about the disk that contains the current working directory are returned.

Julia 1.8

This method was added in Julia 1.8.

source

`Base.Filesystem.lstat` – Function.

```
lstat(path)
lstat(path_elements...)
```

Like `stat`, but for symbolic links gets the info for the link itself rather than the file it refers to.

This function must be called on a file path rather than a file object or a file descriptor.

source

`Base.Filesystem.ctime` – Function.

```
ctime(path)
ctime(path_elements...)
ctime(stat_struct)
```

Return the unix timestamp of when the metadata of the file at `path` was last modified, or the last modified metadata timestamp indicated by the file descriptor `stat_struct`.

Equivalent to `stat(path).ctime` or `stat_struct.ctime`.

source

`Base.Filesystem.mtime` – Function.

```
mtime(path)
mtime(path_elements...)
mtime(stat_struct)
```

Return the unix timestamp of when the file at `path` was last modified, or the last modified timestamp indicated by the file descriptor `stat_struct`.

Equivalent to `stat(path).mtime` or `stat_struct.mtime`.

source

`Base.Filesystem.filemode` – Function.

```
filemode(path)
filemode(path_elements...)
filemode(stat_struct)
```

Return the mode of the file located at `path`, or the mode indicated by the file descriptor `stat_struct`.

Equivalent to `stat(path).mode` or `stat_struct.mode`.

source

Base.filesize – Function.

```
filesize(path)
filesize(path_elements...)
filesize(stat_struct)
```

Return the size of the file located at path, or the size indicated by file descriptor stat_struct.

Equivalent to stat(path).size or stat_struct.size.

source

Base.Filesystem.uperm – Function.

```
uperm(path)
uperm(path_elements...)
uperm(stat_struct)
```

Return a bitfield of the owner permissions for the file at path or file descriptor stat_struct.

Value	Description
01	Execute Permission
02	Write Permission
04	Read Permission

The fact that a bitfield is returned means that if the permission is read+write, the bitfield is "110", which maps to the decimal value of 0+2+4=6. This is reflected in the printing of the returned UInt8 value.

See also gperm and operm.

```
julia> touch("dummy_file");  # Create test-file without contents

julia> uperm("dummy_file")
0x06

julia> bitstring(ans)
"00000110"

julia> has_read_permission(path) = uperm(path) & 0b00000100 != 0;  # Use bit mask to check
↪  specific bit

julia> has_read_permission("dummy_file")
true

julia> rm("dummy_file")     # Clean up test-file
```

source

Base.Filesystem.gperm – Function.

```
gperm(path)
gperm(path_elements...)
gperm(stat_struct)
```

Like uperm but gets the permissions of the group owning the file.

See also operm.

source

Base.Filesystem.operm – Function.

```
operm(path)
operm(path_elements...)
operm(stat_struct)
```

Like uperm but gets the permissions for people who neither own the file nor are a member of the group owning the file.

See also gperm.

source

Base.Filesystem.cp – Function.

```
cp(src::AbstractString, dst::AbstractString; force::Bool=false, follow_symlinks::Bool=false)
```

Copy the file, link, or directory from src to dst. force=true will first remove an existing dst.

If follow_symlinks=false, and src is a symbolic link, dst will be created as a symbolic link. If follow_symlinks=true and src is a symbolic link, dst will be a copy of the file or directory src refers to. Return dst.

Note

The cp function is different from the cp Unix command. The cp function always operates on the assumption that dst is a file, while the command does different things depending on whether dst is a directory or a file. Using force=true when dst is a directory will result in loss of all the contents present in the dst directory, and dst will become a file that has the contents of src instead.

source

Base.download – Function.

```
download(url::AbstractString, [path::AbstractString = tempname()]) -> path
```

Download a file from the given url, saving it to the location path, or if not specified, a temporary path. Returns the path of the downloaded file.

Note

Since Julia 1.6, this function is deprecated and is just a thin wrapper around Downloads.download. In new code, you should use that function directly instead of calling this.

source

Base.Filesystem.mv – Function.

```
mv(src::AbstractString, dst::AbstractString; force::Bool=false)
```

Move the file, link, or directory from `src` to `dst`. `force=true` will first remove an existing `dst`. Return `dst`.

Examples

```
julia> write("hello.txt", "world");

julia> mv("hello.txt", "goodbye.txt")
"goodbye.txt"

julia> "hello.txt" in readdir()
false

julia> readline("goodbye.txt")
"world"

julia> write("hello.txt", "world2");

julia> mv("hello.txt", "goodbye.txt")
ERROR: ArgumentError: 'goodbye.txt' exists. `force=true` is required to remove 'goodbye.txt'
↪  before moving.
Stacktrace:
 [1] #checkfor_mv_cp_cptree#10(::Bool, ::Function, ::String, ::String, ::String) at
 ↪  ./file.jl:293
[...]

julia> mv("hello.txt", "goodbye.txt", force=true)
"goodbye.txt"

julia> rm("goodbye.txt");
```

Note

The `mv` function is different from the `mv` Unix command. The `mv` function by default will error if `dst` exists, while the command will delete an existing `dst` file by default. Also the `mv` function always operates on the assumption that `dst` is a file, while the command does different things depending on whether `dst` is a directory or a file. Using `force=true` when `dst` is a directory will result in loss of all the contents present in the `dst` directory, and `dst` will become a file that has the contents of `src` instead.

source

`Base.Filesystem.rename` – Function.

```
Base.rename(oldpath::AbstractString, newpath::AbstractString)
```

Change the name of a file or directory from `oldpath` to `newpath`. If `newpath` is an existing file or empty directory it may be replaced. Equivalent to rename(2) on Unix. If a path contains a "\0" throw an `ArgumentError`. On other failures throw an `IOError`. Return `newpath`.

This is a lower level filesystem operation used to implement `mv`.

OS-specific restrictions may apply when `oldpath` and `newpath` are in different directories.

Currently there are a few differences in behavior on Windows which may be resolved in a future release. Specifically, currently on Windows:

1. `rename` will fail if `oldpath` or `newpath` are opened files.
2. `rename` will fail if `newpath` is an existing directory.
3. `rename` may work if `newpath` is a file and `oldpath` is a directory.
4. `rename` may remove `oldpath` if it is a hardlink to `newpath`.

See also: `mv`.

Julia 1.12

This method was made public in Julia 1.12.

source

`Base.Filesystem.rm` – Function.

```
rm(path::AbstractString; force::Bool=false, recursive::Bool=false)
```

Delete the file, link, or empty directory at the given path. If `force=true` is passed, a non-existing path is not treated as error. If `recursive=true` is passed and the path is a directory, then all contents are removed recursively.

Examples

```
julia> mkpath("my/test/dir");

julia> rm("my", recursive=true)

julia> rm("this_file_does_not_exist", force=true)

julia> rm("this_file_does_not_exist")
ERROR: IOError: unlink("this_file_does_not_exist"): no such file or directory (ENOENT)
Stacktrace:
[...]
```

source

`Base.Filesystem.touch` – Function.

```
Base.touch(::Pidfile.LockMonitor)
```

Update the `mtime` on the lock, to indicate it is still fresh.

See also the `refresh` keyword in the `mkpidlock` constructor.

```
touch(path::AbstractString)
touch(fd::File)
```

Update the last-modified timestamp on a file to the current time.

If the file does not exist a new file is created.

Return `path`.

Examples

```
julia> write("my_little_file", 2);

julia> mtime("my_little_file")
1.5273815391135583e9

julia> touch("my_little_file");

julia> mtime("my_little_file")
1.527381559163435e9
```

We can see the `mtime` has been modified by `touch`.

source

`Base.Filesystem.tempname` – Function.

```
tempname(parent=tempdir(); cleanup=true, suffix="")::String
```

Generate a temporary file path. This function only returns a path; no file is created. The path is likely to be unique, but this cannot be guaranteed due to the very remote possibility of two simultaneous calls to `tempname` generating the same file name. The name is guaranteed to differ from all files already existing at the time of the call to `tempname`.

When called with no arguments, the temporary name will be an absolute path to a temporary name in the system temporary directory as given by `tempdir()`. If a `parent` directory argument is given, the temporary path will be in that directory instead. If a suffix is given the tempname will end with that suffix and be tested for uniqueness with that suffix.

The `cleanup` option controls whether the process attempts to delete the returned path automatically when the process exits. Note that the `tempname` function does not create any file or directory at the returned location, so there is nothing to cleanup unless you create a file or directory there. If you do and `cleanup` is `true` it will be deleted upon process termination.

Julia 1.4

The `parent` and `cleanup` arguments were added in 1.4. Prior to Julia 1.4 the path `tempname` would never be cleaned up at process termination.

Julia 1.12

The `suffix` keyword argument was added in Julia 1.12.

Warning

This can lead to security holes if another process obtains the same file name and creates the file before you are able to. Open the file with `JL_O_EXCL` if this is a concern. Using `mktemp()` is also recommended instead.

source

Base.Filesystem.tempdir – Function.

```
tempdir()
```

Gets the path of the temporary directory. On Windows, `tempdir()` uses the first environment variable found in the ordered list `TMP`, `TEMP`, `USERPROFILE`. On all other operating systems, `tempdir()` uses the first environment variable found in the ordered list `TMPDIR`, `TMP`, `TEMP`, and `TEMPDIR`. If none of these are found, the path `"/tmp"` is used.

source

Base.Filesystem.mktemp – Method.

```
mktemp(parent=tempdir(); cleanup=true) -> (path, io)
```

Return `(path, io)`, where `path` is the path of a new temporary file in `parent` and `io` is an open file object for this path. The `cleanup` option controls whether the temporary file is automatically deleted when the process exits.

Julia 1.3

The `cleanup` keyword argument was added in Julia 1.3. Relatedly, starting from 1.3, Julia will remove the temporary paths created by `mktemp` when the Julia process exits, unless `cleanup` is explicitly set to `false`.

source

Base.Filesystem.mktemp – Method.

```
mktemp(f::Function, parent=tempdir())
```

Apply the function `f` to the result of `mktemp(parent)` and remove the temporary file upon completion.

See also: `mktempdir`.

source

Base.Filesystem.mktempdir – Method.

```
mktempdir(parent=tempdir(); prefix="jl_", cleanup=true) -> path
```

Create a temporary directory in the `parent` directory with a name constructed from the given `prefix` and a random suffix, and return its path. Additionally, on some platforms, any trailing `'X'` characters in `prefix` may be replaced with random characters. If `parent` does not exist, throw an error. The `cleanup` option controls whether the temporary directory is automatically deleted when the process exits.

Julia 1.2

The `prefix` keyword argument was added in Julia 1.2.

Julia 1.3

The `cleanup` keyword argument was added in Julia 1.3. Relatedly, starting from 1.3, Julia will remove the temporary paths created by `mktempdir` when the Julia process exits, unless `cleanup` is explicitly set to `false`.

See also: `mktemp`, `mkdir`.

source

`Base.Filesystem.mktempdir` – Method.

```
mktempdir(f::Function, parent=tempdir(); prefix="jl_")
```

Apply the function `f` to the result of `mktempdir(parent; prefix)` and remove the temporary directory and all of its contents upon completion.

See also: `mktemp`, `mkdir`.

Julia 1.2

The `prefix` keyword argument was added in Julia 1.2.

source

`Base.Filesystem.isblockdev` – Function.

```
isblockdev(path)::Bool
isblockdev(path_elements...)::Bool
isblockdev(stat_struct)::Bool
```

Return `true` if the path `path` or file descriptor `stat_struct` refer to a block device, `false` otherwise.

source

`Base.Filesystem.ischardev` – Function.

```
ischardev(path)::Bool
ischardev(path_elements...)::Bool
ischardev(stat_struct)::Bool
```

Return `true` if the path `path` or file descriptor `stat_struct` refer to a character device, `false` otherwise.

source

`Base.Filesystem.isdir` – Function.

```
isdir(path)::Bool
isdir(path_elements...)::Bool
```

Return `true` if `path` points to a directory, `false` otherwise.

Examples

```
julia> isdir(homedir())
true

julia> isdir("not/a/directory")
false
```

See also `isfile` and `ispath`.

source

`Base.Filesystem.isfifo` – Function.

```
isfifo(path)::Bool
isfifo(path_elements...)::Bool
isfifo(stat_struct)::Bool
```

Return `true` if the file at `path` or file descriptor `stat_struct` is FIFO, `false` otherwise.

source

`Base.Filesystem.isfile` – Function.

```
isfile(path)::Bool
isfile(path_elements...)::Bool
```

Return `true` if `path` points to a regular file, `false` otherwise.

Examples

```
julia> isfile(homedir())
false

julia> filename = "test_file.txt";

julia> write(filename, "Hello world!");

julia> isfile(filename)
true

julia> rm(filename);

julia> isfile(filename)
false
```

See also `isdir` and `ispath`.

source

`Base.Filesystem.islink` – Function.

```
islink(path)::Bool
islink(path_elements...)::Bool
```

Return `true` if `path` points to a symbolic link, `false` otherwise.

source

Base.Filesystem.ismount – Function.

```
ismount(path)::Bool
ismount(path_elements...)::Bool
```

Return `true` if `path` is a mount point, `false` otherwise.

source

Base.Filesystem.ispath – Function.

```
ispath(path)::Bool
ispath(path_elements...)::Bool
```

Return `true` if a valid filesystem entity exists at `path`, otherwise returns `false`.

This is the generalization of `isfile`, `isdir` etc.

source

Base.Filesystem.issetgid – Function.

```
issetgid(path)::Bool
issetgid(path_elements...)::Bool
issetgid(stat_struct)::Bool
```

Return `true` if the file at `path` or file descriptor `stat_struct` have the setgid flag set, `false` otherwise.

source

Base.Filesystem.issetuid – Function.

```
issetuid(path)::Bool
issetuid(path_elements...)::Bool
issetuid(stat_struct)::Bool
```

Return `true` if the file at `path` or file descriptor `stat_struct` have the setuid flag set, `false` otherwise.

source

Base.Filesystem.issocket – Function.

```
issocket(path)::Bool
issocket(path_elements...)::Bool
```

Return `true` if `path` points to a socket, `false` otherwise.

source

Base.Filesystem.issticky – Function.

```
issticky(path)::Bool
issticky(path_elements...)::Bool
issticky(stat_struct)::Bool
```

Return `true` if the file at `path` or file descriptor `stat_struct` have the sticky bit set, `false` otherwise.

source

Base.Filesystem.homedir – Function.

```
homedir()::String
```

Return the current user's home directory.

Note

homedir determines the home directory via libuv's uv_os_homedir. For details (for example on how to specify the home directory via environment variables), see the uv_os_homedir documentation.

See also Sys.username.

source

Base.Filesystem.dirname – Function.

```
dirname(path::AbstractString)::String
```

Get the directory part of a path. Trailing characters ('/' or '\') in the path are counted as part of the path.

Examples

```
julia> dirname("/home/myuser")
"/home"

julia> dirname("/home/myuser/")
"/home/myuser"
```

See also basename.

source

Base.Filesystem.basename – Function.

```
basename(path::AbstractString)::String
```

Get the file name part of a path.

Note

This function differs slightly from the Unix basename program, where trailing slashes are ignored, i.e. `$ basename /foo/bar/` returns bar, whereas basename in Julia returns an empty string "".

Examples

```
julia> basename("/home/myuser/example.jl")
"example.jl"

julia> basename("/home/myuser/")
""
```

See also dirname.

source

Base.Filesystem.isabspath – Function.

```
isabspath(path::AbstractString)::Bool
```

Determine whether a path is absolute (begins at the root directory).

Examples

```
julia> isabspath("/home")
true

julia> isabspath("home")
false
```

source

Base.Filesystem.isdirpath – Function.

```
isdirpath(path::AbstractString)::Bool
```

Determine whether a path refers to a directory (for example, ends with a path separator).

Examples

```
julia> isdirpath("/home")
false

julia> isdirpath("/home/")
true
```

source

Base.Filesystem.joinpath – Function.

```
joinpath(parts::AbstractString...)::String
joinpath(parts::Vector{AbstractString})::String
joinpath(parts::Tuple{AbstractString})::String
```

Join path components into a full path. If some argument is an absolute path or (on Windows) has a drive specification that doesn't match the drive computed for the join of the preceding paths, then prior components are dropped.

Note on Windows since there is a current directory for each drive, `joinpath("c:", "foo")` represents a path relative to the current directory on drive "c:" so this is equal to "c:foo", not "c:\foo". Furthermore, `joinpath` treats this as a non-absolute path and ignores the drive letter casing, hence `joinpath("C:\\A","c:b") = "C:\\A\\b"`.

Examples

```
julia> joinpath("/home/myuser", "example.jl")
"/home/myuser/example.jl"
```

```
julia> joinpath(["/home/myuser", "example.jl"])
"/home/myuser/example.jl"
```

source

Base.Filesystem.abspath – Function.

```
abspath(path::AbstractString, paths::AbstractString...)::String
```

Convert a set of paths to an absolute path by joining them together and adding the current directory if necessary. Equivalent to `abspath(joinpath(path, paths...))`.

source

```
abspath(path::AbstractString)::String
```

Convert a path to an absolute path by adding the current directory if necessary. Also normalizes the path as in `normpath`.

Examples

If you are in a directory called `JuliaExample` and the data you are using is two levels up relative to the `JuliaExample` directory, you could write:

```
abspath("../../data")
```

Which gives a path like `"/home/JuliaUser/data/"`.

See also `joinpath`, `pwd`, `expanduser`.

source

Base.Filesystem.normpath – Function.

```
normpath(path::AbstractString, paths::AbstractString...)::String
```

Convert a set of paths to a normalized path by joining them together and removing "." and ".." entries. Equivalent to `normpath(joinpath(path, paths...))`.

source

```
normpath(path::AbstractString)::String
```

Normalize a path, removing "." and ".." entries and changing "/" to the canonical path separator for the system.

Examples

```
julia> normpath("/home/myuser/../example.jl")
"/home/example.jl"

julia> normpath("Documents/Julia") == joinpath("Documents", "Julia")
true
```

source

Base.Filesystem.realpath – Function.

```
realpath(path::AbstractString)::String
```

Canonicalize a path by expanding symbolic links and removing "." and ".." entries. On case-insensitive case-preserving filesystems (typically Mac and Windows), the filesystem's stored case for the path is returned.

(This function throws an exception if path does not exist in the filesystem.)

source

Base.Filesystem.relpath – Function.

```
relpath(path::AbstractString, startpath::AbstractString = "."):::String
```

Return a relative filepath to path either from the current directory or from an optional start directory. This is a path computation: the filesystem is not accessed to confirm the existence or nature of path or startpath.

On Windows, case sensitivity is applied to every part of the path except drive letters. If path and startpath refer to different drives, the absolute path of path is returned.

source

Base.Filesystem.expanduser – Function.

```
expanduser(path::AbstractString)::AbstractString
```

On Unix systems, replace a tilde character at the start of a path with the current user's home directory.

See also: contractuser.

source

Base.Filesystem.contractuser – Function.

```
contractuser(path::AbstractString)::AbstractString
```

On Unix systems, if the path starts with homedir(), replace it with a tilde character.

See also: expanduser.

source

Base.Filesystem.samefile – Function.

```
samefile(path_a, path_b)
```

Check if the paths path_a and path_b refer to the same existing file or directory.

source

Base.Filesystem.splitdir – Function.

```
splitdir(path::AbstractString) -> (dir::AbstractString, file::AbstractString)
```

Split a path into a tuple of the directory name and file name.

Examples

```
julia> splitdir("/home/myuser")
("/home", "myuser")
```

source

Base.Filesystem.splitdrive – Function.

```
splitdrive(path::AbstractString) -> (drive::AbstractString, path::AbstractString)
```

On Windows, split a path into the drive letter part and the path part. On Unix systems, the first component is always the empty string.

source

Base.Filesystem.splitext – Function.

```
splitext(path::AbstractString) -> (path_without_extension::String, extension::String)
```

If the last component of a path contains one or more dots, split the path into everything before the last dot and everything including and after the dot. Otherwise, return a tuple of the argument unmodified and the empty string. "splitext" is short for "split extension".

Examples

```
julia> splitext("/home/myuser/example.jl")
("/home/myuser/example", ".jl")

julia> splitext("/home/myuser/example.tar.gz")
("/home/myuser/example.tar", ".gz")

julia> splitext("/home/my.user/example")
("/home/my.user/example", "")
```

source

Base.Filesystem.splitpath – Function.

```
splitpath(path::AbstractString)::Vector{String}
```

Split a file path into all its path components. This is the opposite of `joinpath`. Returns an array of substrings, one for each directory or file in the path, including the root directory if present.

Julia 1.1

This function requires at least Julia 1.1.

Examples

```
julia> splitpath("/home/myuser/example.jl")
4-element Vector{String}:
 "/"
 "home"
 "myuser"
 "example.jl"
```

source

Chapter 55

I/O and Network

55.1 General I/O

`Core.IO` – Type.

```
IO
```

Abstract supertype for input/output types.

source

`Base.stdout` – Constant.

```
stdout::IO
```

Global variable referring to the standard out stream.

source

`Base.stderr` – Constant.

```
stderr::IO
```

Global variable referring to the standard error stream.

source

`Base.stdin` – Constant.

```
stdin::IO
```

Global variable referring to the standard input stream.

source

`Base.read` – Method.

```
read(filename::AbstractString)
```

Read the entire contents of a file as a `Vector{UInt8}`.

```
read(filename::AbstractString, String)
```

Read the entire contents of a file as a string.

```
read(filename::AbstractString, args...)
```

Open a file and read its contents. `args` is passed to `read`: this is equivalent to `open(io->read(io, args...), filename)`.

source

Base.write – Method.

```
write(filename::AbstractString, content)
```

Write the canonical binary representation of `content` to a file, which will be created if it does not exist yet or overwritten if it does exist.

Return the number of bytes written into the file.

source

Base.open – Function.

```
open(f::Function, command, args...; kwargs...)
```

Similar to `open(command, args...; kwargs...)`, but calls `f(stream)` on the resulting process stream, then closes the input stream and waits for the process to complete. Return the value returned by `f` on success. Throw an error if the process failed, or if the process attempts to print anything to stdout.

source

```
open(command, stdio=devnull; write::Bool = false, read::Bool = !write)
```

Start running `command` asynchronously, and return a `process::IO` object. If `read` is true, then reads from the process come from the process's standard output and `stdio` optionally specifies the process's standard input stream. If `write` is true, then writes go to the process's standard input and `stdio` optionally specifies the process's standard output stream. The process's standard error stream is connected to the current global `stderr`.

source

```
open(command, mode::AbstractString, stdio=devnull)
```

Run `command` asynchronously. Like `open(command, stdio; read, write)` except specifying the read and write flags via a mode string instead of keyword arguments. Possible mode strings are:

Mode	Description	Keywords
r	read	none
w	write	write = true
r+	read, write	read = true, write = true
w+	read, write	read = true, write = true

source

```
open(fd::OS_HANDLE)::IO
```

Take a raw file descriptor and wrap it in a Julia-aware IO type, and take ownership of the fd handle. Call `open(Libc.dup(fd))` to avoid the ownership capture of the original handle.

Warning

Do not call this on a handle that's already owned by some other part of the system.

source

```
open(filename::AbstractString, [mode::AbstractString]; lock = true)::IOStream
```

Alternate syntax for open, where a string-based mode specifier is used instead of the five booleans. The values of `mode` correspond to those from `fopen(3)` or Perl `open`, and are equivalent to setting the following boolean groups:

Mode	Description	Keywords
r	read	none
w	write, create, truncate	`write = true`
a	write, create, append	`append = true`
r+	read, write	`read = true, write = true`
w+	read, write, create, truncate	`truncate = true, read = true`
a+	read, write, create, append	`append = true, read = true`

The `lock` keyword argument controls whether operations will be locked for safe multi-threaded access.

Examples

```
julia> io = open("myfile.txt", "w");

julia> write(io, "Hello world!");

julia> close(io);

julia> io = open("myfile.txt", "r");

julia> read(io, String)
"Hello world!"

julia> write(io, "This file is read only")
ERROR: ArgumentError: write failed, IOStream is not writeable
[...]

julia> close(io)

julia> io = open("myfile.txt", "a");

julia> write(io, "This stream is not read only")
28
```

```
julia> close(io)

julia> rm("myfile.txt")
```

Julia 1.5

The `lock` argument is available as of Julia 1.5.

source

```
open(filename::AbstractString; lock = true, keywords...)::IOStream
```

Open a file in a mode specified by five boolean keyword arguments:

Keyword	Description	Default
`read`	open for reading	`!write`
`write`	open for writing	`truncate | append`
`create`	create if non-existent	`!read & write | truncate | append`
`truncate`	truncate to zero size	`!read & write`
`append`	seek to end	`false`

The default when no keywords are passed is to open files for reading only. Returns a stream for accessing the opened file.

The `lock` keyword argument controls whether operations will be locked for safe multi-threaded access.

Julia 1.5

The `lock` argument is available as of Julia 1.5.

source

```
open(f::Function, args...; kwargs...)
```

Apply the function `f` to the result of `open(args...; kwargs...)` and close the resulting file descriptor upon completion.

Examples

```
julia> write("myfile.txt", "Hello world!");

julia> open(io->read(io, String), "myfile.txt")
"Hello world!"

julia> rm("myfile.txt")
```

source

`Base.IOStream` – Type.

```
IOStream
```

A buffered IO stream wrapping an OS file descriptor. Mostly used to represent files returned by open.

source

Base.IOBuffer – Type.

```
IOBuffer(string::String)
```

Create a read-only IOBuffer on the data underlying the given string.

Examples

```
julia> io = IOBuffer("Haho");

julia> takestring!(io)
"Haho"

julia> takestring!(io)
"Haho"
```

source

```
IOBuffer([data::AbstractVector{UInt8}]; keywords...)::IOBuffer
```

Create an in-memory I/O stream, which may optionally operate on a pre-existing array.

It may take optional keyword arguments:

- read, write, append: restricts operations to the buffer; see open for details.
- truncate: truncates the buffer size to zero length.
- maxsize: specifies a size beyond which the buffer may not be grown.
- sizehint: suggests a capacity of the buffer (data must implement sizehint!(data, size)).

When data is not given, the buffer will be both readable and writable by default.

Passing ‘data‘ as scratch space to ‘IOBuffer‘ with ‘write=true‘ may give unexpected behavior

Once write is called on an IOBuffer, it is best to consider any previous references to data invalidated; in effect IOBuffer "owns" this data until a call to take!. Any indirect mutations to data could lead to undefined behavior by breaking the abstractions expected by IOBuffer. If write=true the IOBuffer may store data at any offset leaving behind arbitrary values at other offsets. If maxsize > length(data), the IOBuffer might re-allocate the data entirely, which may or may not be visible in any outstanding bindings to array.

Examples

```
julia> io = IOBuffer();

julia> write(io, "JuliaLang is a GitHub organization.", " It has many members.")
56
```

```
julia> takestring!(io)
"JuliaLang is a GitHub organization. It has many members."

julia> io = IOBuffer(b"JuliaLang is a GitHub organization.")
IOBuffer(data=UInt8[...], readable=true, writable=false, seekable=true, append=false, size=35,
↪  maxsize=Inf, ptr=1, mark=-1)

julia> read(io, String)
"JuliaLang is a GitHub organization."

julia> write(io, "This isn't writable.")
ERROR: ArgumentError: ensureroom failed, IOBuffer is not writeable

julia> io = IOBuffer(UInt8[], read=true, write=true, maxsize=34)
IOBuffer(data=UInt8[...], readable=true, writable=true, seekable=true, append=false, size=0,
↪  maxsize=34, ptr=1, mark=-1)

julia> write(io, "JuliaLang is a GitHub organization.")
34

julia> takestring!(io)
"JuliaLang is a GitHub organization"

julia> length(read(IOBuffer(b"data", read=true, truncate=false)))
4

julia> length(read(IOBuffer(b"data", read=true, truncate=true)))
0
```

source

Base.take! – Method.

```
take!(b::IOBuffer)
```

Obtain the contents of an `IOBuffer` as an array. Afterwards, the `IOBuffer` is reset to its initial state.

Examples

```
julia> io = IOBuffer();

julia> write(io, "JuliaLang is a GitHub organization.", " It has many members.")
56

julia> String(take!(io))
"JuliaLang is a GitHub organization. It has many members."
```

source

Base.Pipe – Type.

```
Pipe()
```

Construct an uninitialized Pipe object, especially for IO communication between multiple processes.

The appropriate end of the pipe will be automatically initialized if the object is used in process spawning. This can be useful to easily obtain references in process pipelines, e.g.:

```
julia> err = Pipe()

# After this `err` will be initialized and you may read `foo`'s
# stderr from the `err` pipe, or pass `err` to other pipelines.
julia> run(pipeline(pipeline(`foo`, stderr=err), `cat`), wait=false)

# Now destroy the write half of the pipe, so that the read half will get EOF
julia> closewrite(err)

julia> read(err, String)
"stderr messages"
```

See also Base.link_pipe!.

source

Base.link_pipe! – Function.

```
link_pipe!(pipe; reader_supports_async=false, writer_supports_async=false)
```

Initialize pipe and link the in endpoint to the out endpoint. The keyword arguments reader_supports_async/writer_support correspond to OVERLAPPED on Windows and O_NONBLOCK on POSIX systems. They should be true unless they'll be used by an external program (e.g. the output of a command executed with run).

source

Base.fdio – Function.

```
fdio([name::AbstractString, ]fd::Integer[, own::Bool=false])::IOStream
```

Create an IOStream object from an integer file descriptor. If own is true, closing this object will close the underlying descriptor. By default, an IOStream is closed when it is garbage collected. name allows you to associate the descriptor with a named file.

source

Base.flush – Function.

```
flush(io::IO)
```

Commit all currently buffered writes to the given io. This has a default implementation flush(::IO) = nothing, so may be called in generic IO code.

source

Base.close – Function.

```
close(io::IO)
```

Close `io`. Performs a `flush` first.

Closing an IO signals that its underlying resources (OS handle, network connections, etc) should be destroyed. A closed IO is in an undefined state and should not be written to or read from. When attempting to do so, the IO may throw an exception, continue to behave normally, or read/write zero bytes, depending on the implementation. However, implementations should make sure that reading to or writing from a closed IO does not cause undefined behaviour.

See also: `isopen`

source

`Base.closewrite` – Function.

```
closewrite(stream)
```

Shutdown the write half of a full-duplex I/O stream. Performs a `flush` first. Notify the other end that no more data will be written to the underlying file. This is not supported by all IO types.

If implemented, `closewrite` causes subsequent `read` or `eof` calls that would block to instead throw EOF or return true, respectively. If the stream is already closed, this is idempotent.

Examples

```
julia> io = Base.BufferStream(); # this never blocks, so we can read and write on the same
↪  Task

julia> write(io, "request");

julia> # calling `read(io)` here would block forever

julia> closewrite(io);

julia> read(io, String)
"request"
```

source

`Base.write` – Function.

```
write(io::IO, x)
```

Write the canonical binary representation of a value to the given I/O stream or file. Return the number of bytes written into the stream. See also `print` to write a text representation (with an encoding that may depend upon `io`).

The endianness of the written value depends on the endianness of the host system. Convert to/from a fixed endianness when writing/reading (e.g. using `htol` and `ltoh`) to get results that are consistent across platforms.

You can write multiple values with the same `write` call, i.e. the following are equivalent:

```
write(io, x, y...)
write(io, x) + write(io, y...)
```

Examples

Consistent serialization:

```
julia> fname = tempname(); # random temporary filename

julia> open(fname,"w") do f
           # Make sure we write 64bit integer in little-endian byte order
           write(f,htol(Int64(42)))
       end
8

julia> open(fname,"r") do f
           # Convert back to host byte order and host integer type
           Int(ltoh(read(f,Int64)))
       end
42
```

Merging write calls:

```
julia> io = IOBuffer();

julia> write(io, "JuliaLang is a GitHub organization.", " It has many members.")
56

julia> takestring!(io)
"JuliaLang is a GitHub organization. It has many members."

julia> write(io, "Sometimes those members") + write(io, " write documentation.")
44

julia> takestring!(io)
"Sometimes those members write documentation."
```

User-defined plain-data types without `write` methods can be written when wrapped in a `Ref`:

```
julia> struct MyStruct; x::Float64; end

julia> io = IOBuffer()
IOBuffer(data=UInt8[...], readable=true, writable=true, seekable=true, append=false, size=0,
↪  maxsize=Inf, ptr=1, mark=-1)

julia> write(io, Ref(MyStruct(42.0)))
8

julia> seekstart(io); read!(io, Ref(MyStruct(NaN)))
Base.RefValue{MyStruct}(MyStruct(42.0))
```

source

`Base.read` – Function.

```
read(command::Cmd, String)
```

Run `command` and return the resulting output as a `String`.

source

```
read(command::Cmd)
```

Run `command` and return the resulting output as an array of bytes.

source

```
read(s::IOStream, nb::Integer; all=true)
```

Read at most `nb` bytes from `s`, returning a `Vector{UInt8}` of the bytes read.

If `all` is `true` (the default), this function will block repeatedly trying to read all requested bytes, until an error or end-of-file occurs. If `all` is `false`, at most one `read` call is performed, and the amount of data returned is device-dependent. Note that not all stream types support the `all` option.

source

```
read(s::IO, nb=typemax(Int))
```

Read at most `nb` bytes from `s`, returning a `Vector{UInt8}` of the bytes read.

source

```
read(filename::AbstractString)
```

Read the entire contents of a file as a `Vector{UInt8}`.

```
read(filename::AbstractString, String)
```

Read the entire contents of a file as a string.

```
read(filename::AbstractString, args...)
```

Open a file and read its contents. `args` is passed to `read`: this is equivalent to `open(io->read(io, args...), filename)`.

source

```
read(io::IO, T)
```

Read a single value of type `T` from `io`, in canonical binary representation.

Note that Julia does not convert the endianness for you. Use `ntoh` or `ltoh` for this purpose.

```
read(io::IO, String)
```

Read the entirety of `io`, as a `String` (see also `readchomp`).

Examples

```
julia> io = IOBuffer("JuliaLang is a GitHub organization");

julia> read(io, Char)
'J': ASCII/Unicode U+004A (category Lu: Letter, uppercase)

julia> io = IOBuffer("JuliaLang is a GitHub organization");

julia> read(io, String)
"JuliaLang is a GitHub organization"
```

source

Base.read! – Function.

```
read!(stream::IO, array::AbstractArray)
read!(filename::AbstractString, array::AbstractArray)
```

Read binary data from an I/O stream or file, filling in array.

source

Base.readbytes! – Function.

```
readbytes!(stream::IOStream, b::AbstractVector{UInt8}, nb=length(b); all::Bool=true)
```

Read at most nb bytes from stream into b, returning the number of bytes read. The size of b will be increased if needed (i.e. if nb is greater than length(b) and enough bytes could be read), but it will never be decreased.

If all is true (the default), this function will block repeatedly trying to read all requested bytes, until an error or end-of-file occurs. If all is false, at most one read call is performed, and the amount of data returned is device-dependent. Note that not all stream types support the all option.

source

```
readbytes!(stream::IO, b::AbstractVector{UInt8}, nb=length(b))
```

Read at most nb bytes from stream into b, returning the number of bytes read. The size of b will be increased if needed (i.e. if nb is greater than length(b) and enough bytes could be read), but it will never be decreased.

source

Base.unsafe_read – Function.

```
unsafe_read(io::IO, ref, nbytes::UInt)
```

Copy nbytes from the IO stream object into ref (converted to a pointer).

It is recommended that subtypes T<:IO override the following method signature to provide more efficient implementations: unsafe_read(s::T, p::Ptr{UInt8}, n::UInt)

source

Base.unsafe_write – Function.

```
unsafe_write(io::IO, ref, nbytes::UInt)
```

Copy nbytes from ref (converted to a pointer) into the IO object.

It is recommended that subtypes T<:IO override the following method signature to provide more efficient implementations: unsafe_write(s::T, p::Ptr{UInt8}, n::UInt)

source

Base.readeach – Function.

```
readeach(io::IO, T)
```

Return an iterable object yielding read(io, T).

See also skipchars, eachline, readuntil.

Julia 1.6

readeach requires Julia 1.6 or later.

Examples

```
julia> io = IOBuffer("JuliaLang is a GitHub organization.\n It has many members.\n");

julia> for c in readeach(io, Char)
           c == '\n' && break
           print(c)
       end
JuliaLang is a GitHub organization.
```

source

Base.peek – Function.

```
peek(stream[, T=UInt8])
```

Read and return a value of type T from a stream without advancing the current position in the stream. See also startswith(stream, char_or_string).

Examples

```
julia> b = IOBuffer("julia");

julia> peek(b)
0x6a

julia> position(b)
0

julia> peek(b, Char)
'j': ASCII/Unicode U+006A (category Ll: Letter, lowercase)
```

Julia 1.5

The method which accepts a type requires Julia 1.5 or later.

source

Base.position – Function.

```
position(l::Lexer)
```

Returns the current position.

source

```
position(s)
```

Get the current position of a stream.

Examples

```
julia> io = IOBuffer("JuliaLang is a GitHub organization.");

julia> seek(io, 5);

julia> position(io)
5

julia> skip(io, 10);

julia> position(io)
15

julia> seekend(io);

julia> position(io)
35
```

source

Base.seek – Function.

```
seek(s, pos)
```

Seek a stream to the given position.

Examples

```
julia> io = IOBuffer("JuliaLang is a GitHub organization.");

julia> seek(io, 5);

julia> read(io, Char)
'L': ASCII/Unicode U+004C (category Lu: Letter, uppercase)
```

source

Base.seekstart – Function.

```
seekstart(s)
```

Seek a stream to its beginning.

Examples

```
julia> io = IOBuffer("JuliaLang is a GitHub organization.");

julia> seek(io, 5);

julia> read(io, Char)
'L': ASCII/Unicode U+004C (category Lu: Letter, uppercase)

julia> seekstart(io);

julia> read(io, Char)
'J': ASCII/Unicode U+004A (category Lu: Letter, uppercase)
```

source

Base.seekend – Function.

```
seekend(s)
```

Seek a stream to its end.

source

Base.skip – Function.

```
skip(s, offset)
```

Seek a stream relative to the current position.

Examples

```
julia> io = IOBuffer("JuliaLang is a GitHub organization.");

julia> seek(io, 5);

julia> skip(io, 10);

julia> read(io, Char)
'G': ASCII/Unicode U+0047 (category Lu: Letter, uppercase)
```

source

Base.mark – Function.

```
mark(s::IO)
```

Add a mark at the current position of stream `s`. Return the marked position.

See also unmark, reset, ismarked.

source

`Base.unmark` – Function.

```
unmark(s::IO)
```

Remove a mark from stream `s`. Return `true` if the stream was marked, `false` otherwise.

See also `mark`, `reset`, `ismarked`.

source

`Base.reset` – Method.

```
reset(s::IO)
```

Reset a stream `s` to a previously marked position, and remove the mark. Return the previously marked position. Throw an error if the stream is not marked.

See also `mark`, `unmark`, `ismarked`.

source

`Base.ismarked` – Function.

```
ismarked(s::IO)
```

Return `true` if stream `s` is marked.

See also `mark`, `unmark`, `reset`.

source

`Base.eof` – Function.

```
eof(stream)::Bool
```

Test whether an I/O stream is at end-of-file. If the stream is not yet exhausted, this function will block to wait for more data if necessary, and then return `false`. Therefore it is always safe to read one byte after seeing `eof` return `false`. `eof` will return `false` as long as buffered data is still available, even if the remote end of a connection is closed.

Examples

```
julia> b = IOBuffer("my buffer");

julia> eof(b)
false

julia> seekend(b);

julia> eof(b)
true
```

source

`Base.isreadonly` – Function.

```
isreadonly(io)::Bool
```

Determine whether a stream is read-only.

Examples

```
julia> io = IOBuffer("JuliaLang is a GitHub organization");

julia> isreadonly(io)
true

julia> io = IOBuffer();

julia> isreadonly(io)
false
```

source

`Base.iswritable` – Function.

```
iswritable(io)::Bool
```

Return `false` if the specified IO object is not writable.

Examples

```
julia> open("myfile.txt", "w") do io
           print(io, "Hello world!");
           iswritable(io)
       end
true

julia> open("myfile.txt", "r") do io
           iswritable(io)
       end
false

julia> rm("myfile.txt")
```

source

```
iswritable(path::String)
```

Return `true` if the access permissions for the given `path` permitted writing by the current user.

Note

This permission may change before the user calls `open`, so it is recommended to just call `open` alone and handle the error if that fails, rather than calling `iswritable` first.

Note

Currently this function does not correctly interrogate filesystem ACLs on Windows, therefore it can return wrong results.

Julia 1.11

This function requires at least Julia 1.11.

See also `ispath`, `isexecutable`, `isreadable`.

source

`Base.isreadable` – Function.

```
isreadable(io)::Bool
```

Return `false` if the specified IO object is not readable.

Examples

```
julia> open("myfile.txt", "w") do io
           print(io, "Hello world!");
           isreadable(io)
       end
false

julia> open("myfile.txt", "r") do io
           isreadable(io)
       end
true

julia> rm("myfile.txt")
```

source

```
isreadable(path::String)
```

Return `true` if the access permissions for the given `path` permitted reading by the current user.

Note

This permission may change before the user calls open, so it is recommended to just call open alone and handle the error if that fails, rather than calling `isreadable` first.

Note

Currently this function does not correctly interrogate filesystem ACLs on Windows, therefore it can return wrong results.

Julia 1.11

This function requires at least Julia 1.11.

See also `ispath`, `isexecutable`, `iswritable`.

source

`Base.isexecutable` – Function.

```
isexecutable(path::String)
```

Return `true` if the given `path` has executable permissions.

Note

This permission may change before the user executes `path`, so it is recommended to execute the file and handle the error if that fails, rather than calling `isexecutable` first.

Note

Prior to Julia 1.6, this did not correctly interrogate filesystem ACLs on Windows, therefore it would return `true` for any file. From Julia 1.6 on, it correctly determines whether the file is marked as executable or not.

See also `ispath`, `isreadable`, `iswritable`.

source

`Base.isopen` – Function.

```
isopen(object)::Bool
```

Determine whether an object, such as an IO or timer, is still open and hence active.

See also: `close`

Examples

```
julia> io = open("my_file.txt", "w+");

julia> isopen(io)
true

julia> close(io)

julia> isopen(io)
false
```

source

`Base.fd` – Function.

```
fd(x)::RawFD
```

Return the file descriptor backing the stream, file, or socket.

RawFD objects can be passed directly to other languages via the `ccall` interface.

Julia 1.12

Prior to 1.12, this function returned an `Int` instead of a RawFD. You may use `RawFD(fd(x))` to produce a RawFD in all Julia versions.

Julia 1.12

Getting the file descriptor of sockets are supported as of Julia 1.12.

Warning

Duplicate the returned file descriptor with `Libc.dup()` before passing it to another system that will take ownership of it (e.g. a C library). Otherwise both the Julia object x and the other system may try to close the file descriptor, which will cause errors.

Warning

The file descriptors for sockets are asynchronous (i.e. `O_NONBLOCK` on POSIX and `OVERLAPPED` on Windows), they may behave differently than regular file descriptors.

source

`Base.redirect_stdio` – Function.

```
redirect_stdio(f; stdin=nothing, stderr=nothing, stdout=nothing)
```

Redirect a subset of the streams `stdin`, `stderr`, `stdout`, call `f()` and restore each stream.

Possible values for each stream are:

- `nothing` indicating the stream should not be redirected.
- `path::AbstractString` redirecting the stream to the file at `path`.
- `io` an `IOStream`, TTY, `Pipe`, socket, or `devnull`.

Examples

```
julia> redirect_stdio(stdout="stdout.txt", stderr="stderr.txt") do
           print("hello stdout")
           print(stderr, "hello stderr")
       end

julia> read("stdout.txt", String)
"hello stdout"

julia> read("stderr.txt", String)
"hello stderr"
```

Edge cases

It is possible to pass the same argument to `stdout` and `stderr`:

```
julia> redirect_stdio(stdout="log.txt", stderr="log.txt", stdin=devnull) do
    ...
end
```

However it is not supported to pass two distinct descriptors of the same file.

```
julia> io1 = open("same/path", "w")

julia> io2 = open("same/path", "w")

julia> redirect_stdio(f, stdout=io1, stderr=io2) # not supported
```

Also the `stdin` argument may not be the same descriptor as `stdout` or `stderr`.

```
julia> io = open(...)

julia> redirect_stdio(f, stdout=io, stdin=io) # not supported
```

Julia 1.7

`redirect_stdio` requires Julia 1.7 or later.

source

```
redirect_stdio(;stdin=stdin, stderr=stderr, stdout=stdout)
```

Redirect a subset of the streams `stdin`, `stderr`, `stdout`. Each argument must be an `IOStream`, TTY, `Pipe`, socket, or `devnull`.

Julia 1.7

`redirect_stdio` requires Julia 1.7 or later.

source

`Base.redirect_stdout` – Function.

```
redirect_stdout([stream]) -> stream
```

Create a pipe to which all C and Julia level `stdout` output will be redirected. Return a stream representing the pipe ends. Data written to `stdout` may now be read from the `rd` end of the pipe.

Note

`stream` must be a compatible objects, such as an `IOStream`, TTY, `Pipe`, socket, or `devnull`.

See also `redirect_stdio`.

source

`Base.redirect_stdout` – Method.

```
redirect_stdout(f::Function, stream)
```

Run the function `f` while redirecting `stdout` to `stream`. Upon completion, `stdout` is restored to its prior setting.

source

`Base.redirect_stderr` – Function.

```
redirect_stderr([stream]) -> stream
```

Like redirect_stdout, but for stderr.

Note

stream must be a compatible objects, such as an IOStream, TTY, Pipe, socket, or devnull.

See also redirect_stdio.

source

Base.redirect_stderr – Method.

```
redirect_stderr(f::Function, stream)
```

Run the function f while redirecting stderr to stream. Upon completion, stderr is restored to its prior setting.

source

Base.redirect_stdin – Function.

```
redirect_stdin([stream]) -> stream
```

Like redirect_stdout, but for stdin. Note that the direction of the stream is reversed.

Note

stream must be a compatible objects, such as an IOStream, TTY, Pipe, socket, or devnull.

See also redirect_stdio.

source

Base.redirect_stdin – Method.

```
redirect_stdin(f::Function, stream)
```

Run the function f while redirecting stdin to stream. Upon completion, stdin is restored to its prior setting.

source

Base.readchomp – Function.

```
readchomp(x)
```

Read the entirety of x as a string and remove a single trailing newline if there is one. Equivalent to chomp(read(x, String)).

Examples

```
julia> write("my_file.txt", "JuliaLang is a GitHub organization.\nIt has many members.\n");

julia> readchomp("my_file.txt")
"JuliaLang is a GitHub organization.\nIt has many members."

julia> rm("my_file.txt");
```

source

Base.truncate – Function.

```
truncate(file, n)
```

Resize the file or buffer given by the first argument to exactly n bytes, filling previously unallocated space with '\0' if the file or buffer is grown.

Examples

```
julia> io = IOBuffer();

julia> write(io, "JuliaLang is a GitHub organization.")
35

julia> truncate(io, 15)
IOBuffer(data=UInt8[...], readable=true, writable=true, seekable=true, append=false, size=15,
↪  maxsize=Inf, ptr=16, mark=-1)

julia> takestring!(io)
"JuliaLang is a "

julia> io = IOBuffer();

julia> write(io, "JuliaLang is a GitHub organization.");

julia> truncate(io, 40);

julia> takestring!(io)
"JuliaLang is a GitHub organization.\0\0\0\0\0"
```

source

Base.skipchars – Function.

```
skipchars(predicate, io::IO; linecomment=nothing)
```

Advance the stream `io` such that the next-read character will be the first remaining for which `predicate` returns `false`. If the keyword argument `linecomment` is specified, all characters from that character until the start of the next line are ignored.

Examples

```
julia> buf = IOBuffer("    text")
IOBuffer(data=UInt8[...], readable=true, writable=false, seekable=true, append=false, size=8,
↪  maxsize=Inf, ptr=1, mark=-1)

julia> skipchars(isspace, buf)
IOBuffer(data=UInt8[...], readable=true, writable=false, seekable=true, append=false, size=8,
↪  maxsize=Inf, ptr=5, mark=-1)

julia> String(readavailable(buf))
"text"
```

source

Base.countlines – Function.

```
countlines(io::IO; eol::AbstractChar = '\n')
countlines(filename::AbstractString; eol::AbstractChar = '\n')
```

Read io until the end of the stream/file and count the number of lines. To specify a file pass the filename as the first argument. EOL markers other than '\n' are supported by passing them as the second argument. The last non-empty line of io is counted even if it does not end with the EOL, matching the length returned by eachline and readlines.

To count lines of a String, countlines(IOBuffer(str)) can be used.

Examples

```
julia> io = IOBuffer("JuliaLang is a GitHub organization.\n");

julia> countlines(io)
1

julia> io = IOBuffer("JuliaLang is a GitHub organization.");

julia> countlines(io)
1

julia> eof(io) # counting lines moves the file pointer
true

julia> io = IOBuffer("JuliaLang is a GitHub organization.");

julia> countlines(io, eol = '.')
1
```

```
julia> write("my_file.txt", "JuliaLang is a GitHub organization.\n")
36

julia> countlines("my_file.txt")
1

julia> countlines("my_file.txt", eol = 'n')
4

julia> rm("my_file.txt")
```

source

Base.PipeBuffer – Function.

```
PipeBuffer(data::AbstractVector{UInt8}=UInt8[]; maxsize::Integer = typemax(Int))
```

An IOBuffer that allows reading and performs writes by appending. Seeking and truncating are not supported. See IOBuffer for the available constructors. If data is given, creates a PipeBuffer to

operate on a data vector, optionally specifying a size beyond which the underlying `Array` may not be grown.

source

`Base.readavailable` – Function.

```
readavailable(stream)
```

Read available buffered data from a stream. Actual I/O is performed only if no data has already been buffered. The result is a `Vector{UInt8}`.

Warning

The amount of data returned is implementation-dependent; for example it can depend on the internal choice of buffer size. Other functions such as `read` should generally be used instead.

source

`Base.IOContext` – Type.

```
IOContext
```

`IOContext` provides a mechanism for passing output configuration settings among `show` methods.

In short, it is an immutable dictionary that is a subclass of `IO`. It supports standard dictionary operations such as `getindex`, and can also be used as an I/O stream.

source

`Base.IOContext` – Method.

```
IOContext(io::IO, KV::Pair...)
```

Create an `IOContext` that wraps a given stream, adding the specified key=>`value` pairs to the properties of that stream (note that `io` can itself be an `IOContext`).

- use `(key => value) in io` to see if this particular combination is in the properties set
- use `get(io, key, default)` to retrieve the most recent value for a particular key

The following properties are in common use:

- `:compact`: Boolean specifying that values should be printed more compactly, e.g. that numbers should be printed with fewer digits. This is set when printing array elements. `:compact` output should not contain line breaks.
- `:limit`: Boolean specifying that containers should be truncated, e.g. showing … in place of most elements.
- `:displaysize`: A `Tuple{Int,Int}` giving the size in rows and columns to use for text output. This can be used to override the display size for called functions, but to get the size of the screen use the `displaysize` function.

- `:typeinfo`: a Type characterizing the information already printed concerning the type of the object about to be displayed. This is mainly useful when displaying a collection of objects of the same type, so that redundant type information can be avoided (e.g. `[Float16(0)]` can be shown as "Float16[0.0]" instead of "Float16[Float16(0.0)]" : while displaying the elements of the array, the `:typeinfo` property will be set to `Float16`).
- `:color`: Boolean specifying whether ANSI color/escape codes are supported/expected. By default, this is determined by whether `io` is a compatible terminal and by any `--color` command-line flag when `julia` was launched.

Examples

```
julia> io = IOBuffer();

julia> printstyled(IOContext(io, :color => true), "string", color=:red)

julia> takestring!(io)
"\e[31mstring\e[39m"

julia> printstyled(io, "string", color=:red)

julia> takestring!(io)
"string"
```

```
julia> print(IOContext(stdout, :compact => false), 1.12341234)
1.12341234
julia> print(IOContext(stdout, :compact => true), 1.12341234)
1.12341
```

```
julia> function f(io::IO)
           if get(io, :short, false)
               print(io, "short")
           else
               print(io, "looooooong")
           end
       end
f (generic function with 1 method)

julia> f(stdout)
looooooong
julia> f(IOContext(stdout, :short => true))
short
```

source

Base.IOContext – Method.

```
IOContext(io::IO, context::IOContext)
```

Create an `IOContext` that wraps an alternate `IO` but inherits the properties of `context`.

Note

Unless explicitly set in the wrapped `io` the `displaysize` of `io` will not be inherited. This is because by default `displaysize` is not a property of IO objects themselves, but lazily inferred, as the size of the terminal window can change during the lifetime of the IO object.

source

55.2 Text I/O

Base.show – Method.

```
show([io::IO = stdout], x)
```

Write a text representation of a value `x` to the output stream `io`. New types `T` should overload `show(io::IO, x::T)`. The representation used by `show` generally includes Julia-specific formatting and type information, and should be parseable Julia code when possible.

`repr` returns the output of `show` as a string.

For a more verbose human-readable text output for objects of type `T`, define `show(io::IO, ::MIME"text/plain", ::T)` in addition. Checking the `:compact` `IOContext` key (often checked as `get(io, :compact, false)::Bool`) of `io` in such methods is recommended, since some containers show their elements by calling this method with `:compact => true`.

See also `print`, which writes un-decorated representations.

Examples

```
julia> show("Hello World!")
"Hello World!"
julia> print("Hello World!")
Hello World!
```

source

Base.summary – Function.

```
summary(io::IO, x)
str = summary(x)
```

Print to a stream `io`, or return a string `str`, giving a brief description of a value. By default returns `string(typeof(x))`, e.g. `Int64`.

For arrays, returns a string of size and type info, e.g. `10-element Vector{Int64}` or `9×4×5 Array{Float64, 3}`.

Examples

```
julia> summary(1)
"Int64"

julia> summary(zeros(2))
"2-element Vector{Float64}"
```

source

Base.print – Function.

```
print([io::IO], xs...)
```

Write to io (or to the default output stream stdout if io is not given) a canonical (un-decorated) text representation. The representation used by print includes minimal formatting and tries to avoid Julia-specific details.

print falls back to calling the 2-argument show(io, x) for each argument x in xs, so most types should just define show. Define print if your type has a separate "plain" representation. For example, show displays strings with quotes, and print displays strings without quotes.

See also println, string, printstyled.

Examples

```
julia> print("Hello World!")
Hello World!
julia> io = IOBuffer();

julia> print(io, "Hello", ' ', :World!)

julia> takestring!(io)
"Hello World!"
```

source

Base.println – Function.

```
println([io::IO], xs...)
```

Print (using print) xs to io followed by a newline. If io is not supplied, prints to the default output stream stdout.

See also printstyled to add colors etc.

Examples

```
julia> println("Hello, world")
Hello, world

julia> io = IOBuffer();

julia> println(io, "Hello", ',', " world.")

julia> takestring!(io)
"Hello, world.\n"
```

source

Base.printstyled – Function.

```
printstyled([io], xs...; bold::Bool=false, italic::Bool=false, underline::Bool=false,
↪  blink::Bool=false, reverse::Bool=false, hidden::Bool=false,
↪  color::Union{Symbol,Int}=:normal)
```

Print xs in a color specified as a symbol or integer, optionally in bold.

Keyword `color` may take any of the values `:normal`, `:italic`, `:default`, `:bold`, `:black`, `:blink`, `:blue`, `:cyan`, `:green`, `:hidden`, `:light_black`, `:light_blue`, `:light_cyan`, `:light_green`, `:light_magenta`, `:light_red`, `:light_white`, `:light_yellow`, `:magenta`, `:nothing`, `:red`, `:reverse`, `:underline`, `:white`, or `:yellow` or an integer between 0 and 255 inclusive. Note that not all terminals support 256 colors.

Keywords `bold=true`, `italic=true`, `underline=true`, `blink=true` are self-explanatory. Keyword `reverse=true` prints with foreground and background colors exchanged, and `hidden=true` should be invisible in the terminal but can still be copied. These properties can be used in any combination.

See also `print`, `println`, `show`.

Note

Not all terminals support italic output. Some terminals interpret italic as reverse or blink.

Julia 1.7

Keywords except `color` and `bold` were added in Julia 1.7.

Julia 1.10

Support for italic output was added in Julia 1.10.

source

Base.sprint – Function.

```
sprint(f::Function, args...; context=nothing, sizehint=0)
```

Call the given function with an I/O stream and the supplied extra arguments. Everything written to this I/O stream is returned as a string.

The optional keyword argument `context` can be set to a `:key=>value` pair, a tuple of `:key=>value` pairs, or an `IO` or `IOContext` object whose attributes are used for the I/O stream passed to `f`. The optional `sizehint` is a suggested size (in bytes) to allocate for the buffer used to write the string.

Julia 1.7

Passing a tuple to keyword `context` requires Julia 1.7 or later.

Examples

```
julia> sprint(show, 66.66666; context=:compact => true)
"66.6667"

julia> sprint(showerror, BoundsError([1], 100))
"BoundsError: attempt to access 1-element Vector{Int64} at index [100]"
```

source

Base.showerror – Function.

```
showerror(io, e)
```

Show a descriptive representation of an exception object e. This method is used to display the exception after a call to `throw`.

Examples

```
julia> struct MyException <: Exception
           msg::String
       end

julia> function Base.showerror(io::IO, err::MyException)
           print(io, "MyException: ")
           print(io, err.msg)
       end

julia> err = MyException("test exception")
MyException("test exception")

julia> sprint(showerror, err)
"MyException: test exception"

julia> throw(MyException("test exception"))
ERROR: MyException: test exception
```

source

`Base.dump` – Function.

```
dump(x; maxdepth=8)
```

Show every part of the representation of a value. The depth of the output is truncated at `maxdepth`.

Examples

```
julia> struct MyStruct
           x
           y
       end

julia> x = MyStruct(1, (2,3));

julia> dump(x)
MyStruct
  x: Int64 1
  y: Tuple{Int64, Int64}
    1: Int64 2
    2: Int64 3

julia> dump(x; maxdepth = 1)
MyStruct
  x: Int64 1
  y: Tuple{Int64, Int64}
```

source

Base.Meta.@dump – Macro.

```
@dump expr
```

Show every part of the representation of the given expression. Equivalent to dump(:(expr)).

source

Base.readline – Function.

```
readline(io::IO=stdin; keep::Bool=false)
readline(filename::AbstractString; keep::Bool=false)
```

Read a single line of text from the given I/O stream or file (defaults to stdin). When reading from a file, the text is assumed to be encoded in UTF-8. Lines in the input end with '\n' or "\r\n" or the end of an input stream. When keep is false (as it is by default), these trailing newline characters are removed from the line before it is returned. When keep is true, they are returned as part of the line.

Return a String. See also copyline to instead write in-place to another stream (which can be a preallocated IOBuffer).

See also readuntil for reading until more general delimiters.

Examples

```
julia> write("my_file.txt", "JuliaLang is a GitHub organization.\nIt has many members.\n");

julia> readline("my_file.txt")
"JuliaLang is a GitHub organization."

julia> readline("my_file.txt", keep=true)
"JuliaLang is a GitHub organization.\n"

julia> rm("my_file.txt")
```

```
julia> print("Enter your name: ")
Enter your name:

julia> your_name = readline()
Logan
"Logan"
```

source

Base.readuntil – Function.

```
readuntil(stream::IO, delim; keep::Bool = false)
readuntil(filename::AbstractString, delim; keep::Bool = false)
```

Read a string from an I/O stream or a file, up to the given delimiter. The delimiter can be a UInt8, AbstractChar, string, or vector. Keyword argument keep controls whether the delimiter is included in the result. The text is assumed to be encoded in UTF-8.

Return a `String` if `delim` is an `AbstractChar` or a string or otherwise return a `Vector{typeof(delim)}`. See also `copyuntil` to instead write in-place to another stream (which can be a preallocated `IOBuffer`).

Examples

```
julia> write("my_file.txt", "JuliaLang is a GitHub organization.\nIt has many members.\n");

julia> readuntil("my_file.txt", 'L')
"Julia"

julia> readuntil("my_file.txt", '.', keep = true)
"JuliaLang is a GitHub organization."

julia> rm("my_file.txt")
```

source

Base.readlines – Function.

```
readlines(io::IO=stdin; keep::Bool=false)
readlines(filename::AbstractString; keep::Bool=false)
```

Read all lines of an I/O stream or a file as a vector of strings. Behavior is equivalent to saving the result of reading `readline` repeatedly with the same arguments and saving the resulting lines as a vector of strings. See also `eachline` to iterate over the lines without reading them all at once.

Examples

```
julia> write("my_file.txt", "JuliaLang is a GitHub organization.\nIt has many members.\n");

julia> readlines("my_file.txt")
2-element Vector{String}:
 "JuliaLang is a GitHub organization."
 "It has many members."

julia> readlines("my_file.txt", keep=true)
2-element Vector{String}:
 "JuliaLang is a GitHub organization.\n"
 "It has many members.\n"

julia> rm("my_file.txt")
```

source

Base.eachline – Function.

```
eachline(io::IO=stdin; keep::Bool=false)
eachline(filename::AbstractString; keep::Bool=false)
```

Create an iterable `EachLine` object that will yield each line from an I/O stream or a file. Iteration calls `readline` on the stream argument repeatedly with keep passed through, determining whether trailing end-of-line characters are retained. When called with a file name, the file is opened once at the beginning of iteration and closed at the end. If iteration is interrupted, the file will be closed when the `EachLine` object is garbage collected.

To iterate over each line of a `String`, `eachline(IOBuffer(str))` can be used.

`Iterators.reverse` can be used on an EachLine object to read the lines in reverse order (for files, buffers, and other I/O streams supporting `seek`), and `first` or `last` can be used to extract the initial or final lines, respectively.

Examples

```
julia> write("my_file.txt", "JuliaLang is a GitHub organization.\n It has many members.\n");

julia> for line in eachline("my_file.txt")
           print(line)
       end
JuliaLang is a GitHub organization. It has many members.

julia> rm("my_file.txt");
```

Julia 1.8

Julia 1.8 is required to use `Iterators.reverse` or `last` with `eachline` iterators.

source

`Base.copyline` – Function.

```
copyline(out::IO, io::IO=stdin; keep::Bool=false)
copyline(out::IO, filename::AbstractString; keep::Bool=false)
```

Copy a single line of text from an I/O `stream` or a file to the `out` stream, returning `out`.

When reading from a file, the text is assumed to be encoded in UTF-8. Lines in the input end with `'\n'` or `"\r\n"` or the end of an input stream. When keep is false (as it is by default), these trailing newline characters are removed from the line before it is returned. When keep is true, they are returned as part of the line.

Similar to `readline`, which returns a `String`; in contrast, `copyline` writes directly to `out`, without allocating a string. (This can be used, for example, to read data into a pre-allocated `IOBuffer`.)

See also `copyuntil` for reading until more general delimiters.

Examples

```
julia> write("my_file.txt", "JuliaLang is a GitHub organization.\nIt has many members.\n");

julia> takestring!(copyline(IOBuffer(), "my_file.txt"))
"JuliaLang is a GitHub organization."

julia> takestring!(copyline(IOBuffer(), "my_file.txt", keep=true))
"JuliaLang is a GitHub organization.\n"

julia> rm("my_file.txt")
```

Julia 1.11

`copyline` was introduced in Julia 1.11.

source

Base.copyuntil – Function.

```
copyuntil(out::IO, stream::IO, delim; keep::Bool = false)
copyuntil(out::IO, filename::AbstractString, delim; keep::Bool = false)
```

Copy a string from an I/O `stream` or a file, up to the given delimiter, to the `out` stream, returning `out`. The delimiter can be a `UInt8`, `AbstractChar`, string, or vector. Keyword argument keep controls whether the delimiter is included in the result. The text is assumed to be encoded in UTF-8.

Similar to `readuntil`, which returns a `String`; in contrast, `copyuntil` writes directly to `out`, without allocating a string. (This can be used, for example, to read data into a pre-allocated `IOBuffer`.)

Examples

```
julia> write("my_file.txt", "JuliaLang is a GitHub organization.\nIt has many members.\n");

julia> takestring!(copyuntil(IOBuffer(), "my_file.txt", 'L'))
"Julia"

julia> takestring!(copyuntil(IOBuffer(), "my_file.txt", '.', keep = true))
"JuliaLang is a GitHub organization."

julia> rm("my_file.txt")
```

Julia 1.11

`copyuntil` was introduced in Julia 1.11.

source

Base.displaysize – Function.

```
displaysize([io::IO]) -> (lines, columns)
```

Return the nominal size of the screen that may be used for rendering output to this `IO` object. If no input is provided, the environment variables `LINES` and `COLUMNS` are read. If those are not set, a default size of `(24, 80)` is returned.

Examples

```
julia> withenv("LINES" => 30, "COLUMNS" => 100) do
           displaysize()
       end
(30, 100)
```

To get your TTY size,

```
julia> displaysize(stdout)
(34, 147)
```

source

55.3 Multimedia I/O

Just as text output is performed by `print` and user-defined types can indicate their textual representation by overloading `show`, Julia provides a standardized mechanism for rich multimedia output (such as images, formatted text, or even audio and video), consisting of three parts:

- A function `display(x)` to request the richest available multimedia display of a Julia object `x` (with a plain-text fallback).
- Overloading `show` allows one to indicate arbitrary multimedia representations (keyed by standard MIME types) of user-defined types.
- Multimedia-capable display backends may be registered by subclassing a generic `AbstractDisplay` type and pushing them onto a stack of display backends via `pushdisplay`.

The base Julia runtime provides only plain-text display, but richer displays may be enabled by loading external modules or by using graphical Julia environments (such as the IPython-based IJulia notebook).

`Base.Multimedia.AbstractDisplay` – Type.

```
AbstractDisplay
```

Abstract supertype for rich display output devices. `TextDisplay` is a subtype of this.

source

`Base.Multimedia.display` – Function.

```
display(x)
display(d::AbstractDisplay, x)
display(mime, x)
display(d::AbstractDisplay, mime, x)
```

Display `x` using the topmost applicable display in the display stack, typically using the richest supported multimedia output for `x`, with plain-text `stdout` output as a fallback. The `display(d, x)` variant attempts to display `x` on the given display `d` only, throwing a `MethodError` if `d` cannot display objects of this type.

In general, you cannot assume that `display` output goes to `stdout` (unlike `print(x)` or `show(x)`). For example, `display(x)` may open up a separate window with an image. `display(x)` means "show `x` in the best way you can for the current output device(s)." If you want REPL-like text output that is guaranteed to go to `stdout`, use `show(stdout, "text/plain", x)` instead.

There are also two variants with a `mime` argument (a MIME type string, such as `"image/png"`), which attempt to display `x` using the requested MIME type *only*, throwing a `MethodError` if this type is not supported by either the display(s) or by `x`. With these variants, one can also supply the "raw" data in the requested MIME type by passing `x::AbstractString` (for MIME types with text-based storage, such as text/html or application/postscript) or `x::Vector{UInt8}` (for binary MIME types).

To customize how instances of a type are displayed, overload `show` rather than `display`, as explained in the manual section on custom pretty-printing.

source

Base.Multimedia.redisplay – Function.

```
redisplay(x)
redisplay(d::AbstractDisplay, x)
redisplay(mime, x)
redisplay(d::AbstractDisplay, mime, x)
```

By default, the redisplay functions simply call display. However, some display backends may override redisplay to modify an existing display of x (if any). Using redisplay is also a hint to the backend that x may be redisplayed several times, and the backend may choose to defer the display until (for example) the next interactive prompt.

source

Base.Multimedia.displayable – Function.

```
displayable(mime)::Bool
displayable(d::AbstractDisplay, mime)::Bool
```

Return a boolean value indicating whether the given mime type (string) is displayable by any of the displays in the current display stack, or specifically by the display d in the second variant.

source

Base.show – Method.

```
show(io::IO, mime, x)
```

The display functions ultimately call show in order to write an object x as a given mime type to a given I/O stream io (usually a memory buffer), if possible. In order to provide a rich multimedia representation of a user-defined type T, it is only necessary to define a new show method for T, via: show(io, ::MIME"mime", x::T) = ..., where mime is a MIME-type string and the function body calls write (or similar) to write that representation of x to io. (Note that the MIME"" notation only supports literal strings; to construct MIME types in a more flexible manner use MIME{Symbol("")}.)

For example, if you define a MyImage type and know how to write it to a PNG file, you could define a function show(io, ::MIME"image/png", x::MyImage) = ... to allow your images to be displayed on any PNG-capable AbstractDisplay (such as IJulia). As usual, be sure to import Base.show in order to add new methods to the built-in Julia function show.

Technically, the MIME"mime" macro defines a singleton type for the given mime string, which allows us to exploit Julia's dispatch mechanisms in determining how to display objects of any given type.

The default MIME type is MIME"text/plain". There is a fallback definition for text/plain output that calls show with 2 arguments, so it is not always necessary to add a method for that case. If a type benefits from custom human-readable output though, show(::IO, ::MIME"text/plain", ::T) should be defined. For example, the Day type uses 1 day as the output for the text/plain MIME type, and Day(1) as the output of 2-argument show.

Examples

```
julia> struct Day
           n::Int
       end
```

```
julia> Base.show(io::IO, ::MIME"text/plain", d::Day) = print(io, d.n, " day")

julia> Day(1)
1 day
```

Container types generally implement 3-argument `show` by calling `show(io, MIME"text/plain"(), x)` for elements `x`, with `:compact => true` set in an `IOContext` passed as the first argument.

source

`Base.Multimedia.showable` – Function.

```
showable(mime, x)
```

Return a boolean value indicating whether or not the object `x` can be written as the given `mime` type.

(By default, this is determined automatically by the existence of the corresponding `show` method for `typeof(x)`. Some types provide custom `showable` methods; for example, if the available MIME formats depend on the *value* of `x`.)

Examples

```
julia> showable(MIME("text/plain"), rand(5))
true

julia> showable("image/png", rand(5))
false
```

source

`Base.repr` – Method.

```
repr(mime, x; context=nothing)
```

Return an `AbstractString` or `Vector{UInt8}` containing the representation of `x` in the requested `mime` type, as written by `show(io, mime, x)` (throwing a `MethodError` if no appropriate `show` is available). An `AbstractString` is returned for MIME types with textual representations (such as `"text/html"` or `"application/postscript"`), whereas binary data is returned as `Vector{UInt8}`. (The function `istextmime(mime)` returns whether or not Julia treats a given `mime` type as text.)

The optional keyword argument `context` can be set to `:key=>value` pair or an `IO` or `IOContext` object whose attributes are used for the I/O stream passed to `show`.

As a special case, if `x` is an `AbstractString` (for textual MIME types) or a `Vector{UInt8}` (for binary MIME types), the `repr` function assumes that `x` is already in the requested `mime` format and simply returns `x`. This special case does not apply to the `"text/plain"` MIME type. This is useful so that raw data can be passed to `display(m::MIME, x)`.

In particular, `repr("text/plain", x)` is typically a "pretty-printed" version of `x` designed for human consumption. See also `repr(x)` to instead return a string corresponding to `show(x)` that may be closer to how the value of `x` would be entered in Julia.

Examples

```
julia> A = [1 2; 3 4];

julia> repr("text/plain", A)
"2×2 Matrix{Int64}:\n 1  2\n 3  4"
```

source

Base.Multimedia.MIME – Type.

```
MIME
```

A type representing a standard internet data format. "MIME" stands for "Multipurpose Internet Mail Extensions", since the standard was originally used to describe multimedia attachments to email messages.

A MIME object can be passed as the second argument to show to request output in that format.

Examples

```
julia> show(stdout, MIME("text/plain"), "hi")
"hi"
```

source

Base.Multimedia.@MIME_str – Macro.

```
@MIME_str
```

A convenience macro for writing MIME types, typically used when adding methods to show. For example the syntax show(io::IO, ::MIME"text/html", x::MyType) = ... could be used to define how to write an HTML representation of MyType.

source

As mentioned above, one can also define new display backends. For example, a module that can display PNG images in a window can register this capability with Julia, so that calling display(x) on types with PNG representations will automatically display the image using the module's window.

In order to define a new display backend, one should first create a subtype D of the abstract class AbstractDisplay. Then, for each MIME type (mime string) that can be displayed on D, one should define a function display(d::D, ::MIME"mime", x) = ... that displays x as that MIME type, usually by calling show(io, mime, x) or repr(io, mime, x). A MethodError should be thrown if x cannot be displayed as that MIME type; this is automatic if one calls show or repr. Finally, one should define a function display(d::D, x) that queries showable(mime, x) for the mime types supported by D and displays the "best" one; a MethodError should be thrown if no supported MIME types are found for x. Similarly, some subtypes may wish to override redisplay(d::D, ...). (Again, one should import Base.display to add new methods to display.) The return values of these functions are up to the implementation (since in some cases it may be useful to return a display "handle" of some type). The display functions for D can then be called directly, but they can also be invoked automatically from display(x) simply by pushing a new display onto the display-backend stack with:

Base.Multimedia.pushdisplay – Function.

```
pushdisplay(d::AbstractDisplay)
```

Pushes a new display d on top of the global display-backend stack. Calling `display(x)` or `display(mime, x)` will display `x` on the topmost compatible backend in the stack (i.e., the topmost backend that does not throw a `MethodError`).

source

`Base.Multimedia.popdisplay` – Function.

```
popdisplay()
popdisplay(d::AbstractDisplay)
```

Pop the topmost backend off of the display-backend stack, or the topmost copy of `d` in the second variant.

source

`Base.Multimedia.TextDisplay` – Type.

```
TextDisplay(io::IO)
```

Return a `TextDisplay <: AbstractDisplay`, which displays any object as the text/plain MIME type (by default), writing the text representation to the given I/O stream. (This is how objects are printed in the Julia REPL.)

source

`Base.Multimedia.istextmime` – Function.

```
istextmime(m::MIME)
```

Determine whether a MIME type is text data. MIME types are assumed to be binary data except for a set of types known to be text data (possibly Unicode).

Examples

```
julia> istextmime(MIME("text/plain"))
true

julia> istextmime(MIME("image/png"))
false
```

source

55.4 Network I/O

`Base.bytesavailable` – Function.

```
bytesavailable(io)
```

Return the number of bytes available for reading before a read from this stream or buffer will block.

Examples

```
julia> io = IOBuffer("JuliaLang is a GitHub organization");

julia> bytesavailable(io)
34
```

source

Base.ntoh – Function.

```
ntoh(x)
```

Convert the endianness of a value from Network byte order (big-endian) to that used by the Host.

source

Base.hton – Function.

```
hton(x)
```

Convert the endianness of a value from that used by the Host to Network byte order (big-endian).

source

Base.ltoh – Function.

```
ltoh(x)
```

Convert the endianness of a value from Little-endian to that used by the Host.

source

Base.htol – Function.

```
htol(x)
```

Convert the endianness of a value from that used by the Host to Little-endian.

source

Base.ENDIAN_BOM – Constant.

```
ENDIAN_BOM
```

The 32-bit byte-order-mark indicates the native byte order of the host machine. Little-endian machines will contain the value `0x04030201`. Big-endian machines will contain the value `0x01020304`.

source

Chapter 56

Punctuation

Extended documentation for mathematical symbols & functions is here.

sym-bol	meaning
`@`	the at-sign marks a macro invocation; optionally followed by an argument list
`!`	an exclamation mark is a prefix operator for logical negation ("not")
`a!`	function names that end with an exclamation mark modify one or more of their arguments by convention
`#`	the number sign (or hash or pound) character begins single line comments
`#=`	when followed by an equals sign, it begins a multi-line comment (these are nestable)
`=#`	end a multi-line comment by immediately preceding the number sign with an equals sign
`$`	the dollar sign is used for string and expression interpolation
`%`	the percent symbol is the remainder operator
`^`	the caret is the exponentiation operator
`&`	single ampersand is bitwise and
`&&`	double ampersands is short-circuiting boolean and
`|`	single pipe character is bitwise or
`||`	double pipe characters is short-circuiting boolean or
`⊻`	the unicode xor character is bitwise exclusive or
`~`	the tilde is an operator for bitwise not
`'`	a trailing apostrophe is the `adjoint` (that is, the complex transpose) operator A^H
`*`	the asterisk is used for multiplication, including matrix multiplication and string concatenation
`/`	forward slash divides the argument on its left by the one on its right
`//`	double forward slash performs exact, rational division
`\`	backslash operator divides the argument on its right by the one on its left, commonly used to solve matrix equations
`()`	parentheses with no arguments constructs an empty `Tuple`
`(a,...)`	parentheses with comma-separated arguments constructs a tuple containing its arguments
`(a=1,...)`	parentheses with comma-separated assignments constructs a `NamedTuple`
`(x;y)`	parentheses can also be used to group one or more semicolon separated expressions
`a[]`	array indexing (calling `getindex` or `setindex!`)
`[,]`	vector literal constructor (calling `vect`)
`[;]`	vertical concatenation (calling `vcat` or `hvcat`)
`[ ]`	with space-separated expressions, horizontal concatenation (calling `hcat` or `hvcat`)
`T{ }`	curly braces following a type list that type's parameters
`{}`	curly braces can also be used to group multiple `where` expressions in function declarations
`;`	semicolons separate statements, begin a list of keyword arguments in function declarations or calls, or are used to separate array literals for vertical concatenation
`,`	commas separate function arguments or tuple or array components
`?`	the question mark delimits the ternary conditional operator (used like: `conditional ? if_true : if_false`)
`" "`	the single double-quote character delimits `String` literals
`"""` `"""`	three double-quote characters delimits string literals that may contain `"` and ignore leading indentation
`' '`	the single-quote character delimits `Char` (that is, character) literals
`` ` ` ``	the backtick character delimits external process (`Cmd`) literals
`A...`	triple periods are a postfix operator that "splat" their arguments' contents into many arguments of a function call or declare a varargs function that "slurps" up many arguments into a single tuple
`a.b`	single periods access named fields in objects/modules (calling `getproperty` or `setproperty!`)
`f.()`	periods may also prefix parentheses (like `f.(...)`) or infix operators (like `.+`) to perform the function element-wise (calling `broadcast`)
`a:b`	colons (`:`) used as a binary infix operator construct a range from a to b (inclusive) with fixed step size 1
`a:s:b`	colons (`:`) used as a ternary infix operator construct a range from a to b (inclusive) with step size s
`:`	when used by themselves, `Colon`s represent all indices within a dimension, frequently combined with indexing
`::`	double-colons represent a type annotation or `typeassert`, depending on context, frequently used when declaring function arguments
`:( )`	quoted expression
`:a`	`Symbol` a
`<:`	subtype operator

Chapter 57

Sorting and Related Functions

Julia has an extensive, flexible API for sorting and interacting with already-sorted arrays of values. By default, Julia picks reasonable algorithms and sorts in ascending order:

```
julia> sort([2,3,1])
3-element Vector{Int64}:
 1
 2
 3
```

You can sort in reverse order as well:

```
julia> sort([2,3,1], rev=true)
3-element Vector{Int64}:
 3
 2
 1
```

`sort` constructs a sorted copy leaving its input unchanged. Use the "bang" version of the sort function to mutate an existing array:

```
julia> a = [2,3,1];

julia> sort!(a);

julia> a
3-element Vector{Int64}:
 1
 2
 3
```

Instead of directly sorting an array, you can compute a permutation of the array's indices that puts the array into sorted order:

```
julia> v = [0.297288, 0.382396, -0.597634, -0.0104452, -0.839027]
5-element Vector{Float64}:
  0.297288
  0.382396
 -0.597634
 -0.0104452
 -0.839027

julia> p = sortperm(v)
5-element Vector{Int64}:
 5
 3
 4
 1
 2

julia> v[p]
5-element Vector{Float64}:
 -0.839027
 -0.597634
 -0.0104452
  0.297288
  0.382396
```

Arrays can be sorted according to an arbitrary transformation of their values:

```
julia> sort(v, by=abs)
5-element Vector{Float64}:
 -0.0104452
  0.297288
  0.382396
 -0.597634
 -0.839027
```

Or in reverse order by a transformation:

```
julia> sort(v, by=abs, rev=true)
5-element Vector{Float64}:
 -0.839027
 -0.597634
  0.382396
  0.297288
 -0.0104452
```

If needed, the sorting algorithm can be chosen:

```
julia> sort(v, alg=InsertionSort)
5-element Vector{Float64}:
 -0.839027
 -0.597634
 -0.0104452
```

```
0.297288
0.382396
```

All the sorting and order related functions rely on a "less than" relation defining a strict weak order on the values to be manipulated. The `isless` function is invoked by default, but the relation can be specified via the `lt` keyword, a function that takes two array elements and returns `true` if and only if the first argument is "less than" the second. See `sort!` and Alternate Orderings for more information.

57.1 Sorting Functions

`Base.sort!` – Function.

```
sort!(A; dims::Integer, alg::Base.Sort.Algorithm=Base.Sort.defalg(A), lt=isless, by=identity,
↪  rev::Bool=false, order::Base.Order.Ordering=Base.Order.Forward)
```

Sort the multidimensional array A along dimension `dims`. See the one-dimensional version of `sort!` for a description of possible keyword arguments.

To sort slices of an array, refer to `sortslices`.

Julia 1.1

This function requires at least Julia 1.1.

Examples

```
julia> A = [4 3; 1 2]
2×2 Matrix{Int64}:
 4  3
 1  2

julia> sort!(A, dims = 1); A
2×2 Matrix{Int64}:
 1  2
 4  3

julia> sort!(A, dims = 2); A
2×2 Matrix{Int64}:
 1  2
 3  4
```

source

```
sort!(v; alg::Base.Sort.Algorithm=Base.Sort.defalg(v), lt=isless, by=identity,
↪  rev::Bool=false, order::Base.Order.Ordering=Base.Order.Forward)
```

Mutate the vector v so that it is sorted.

A stable algorithm is used by default: the ordering of elements that compare equal is preserved. A specific algorithm can be selected via the `alg` keyword (see Sorting Algorithms for available algorithms).

Elements are first transformed with the function by and then compared according to either the function `lt` or the ordering `order`. Finally, the resulting order is reversed if `rev=true` (this preserves forward

stability: elements that compare equal are not reversed). The current implementation applies the by transformation before each comparison rather than once per element.

Passing an `lt` other than `isless` along with an `order` other than `Base.Order.Forward` or `Base.Order.Reverse` is not permitted, otherwise all options are independent and can be used together in all possible combinations. Note that `order` can also include a "by" transformation, in which case it is applied after that defined with the by keyword. For more information on `order` values see the documentation on Alternate Orderings.

Relations between two elements are defined as follows (with "less" and "greater" exchanged when `rev=true`):

- x is less than y if `lt(by(x), by(y))` (or `Base.Order.lt(order, by(x), by(y))`) yields true.
- x is greater than y if y is less than x.
- x and y are equivalent if neither is less than the other ("incomparable" is sometimes used as a synonym for "equivalent").

The result of `sort!` is sorted in the sense that every element is greater than or equivalent to the previous one.

The `lt` function must define a strict weak order, that is, it must be

- irreflexive: `lt(x, x)` always yields `false`,
- asymmetric: if `lt(x, y)` yields `true` then `lt(y, x)` yields `false`,
- transitive: `lt(x, y) && lt(y, z)` implies `lt(x, z)`,
- transitive in equivalence: `!lt(x, y) && !lt(y, x)` and `!lt(y, z) && !lt(z, y)` together imply `!lt(x, z) && !lt(z, x)`. In words: if x and y are equivalent and y and z are equivalent then x and z must be equivalent.

For example < is a valid `lt` function for `Int` values but ≤ is not: it violates irreflexivity. For `Float64` values even < is invalid as it violates the fourth condition: `1.0` and `NaN` are equivalent and so are `NaN` and `2.0` but `1.0` and `2.0` are not equivalent.

See also `sort`, `sortperm`, `sortslices`, `partialsort!`, `partialsortperm`, `issorted`, `searchsorted`, `insorted`, `Base.Order.ord`.

Examples

```
julia> v = [3, 1, 2]; sort!(v); v
3-element Vector{Int64}:
 1
 2
 3

julia> v = [3, 1, 2]; sort!(v, rev = true); v
3-element Vector{Int64}:
 3
 2
 1

julia> v = [(1, "c"), (3, "a"), (2, "b")]; sort!(v, by = x -> x[1]); v
```

```
3-element Vector{Tuple{Int64, String}}:
 (1, "c")
 (2, "b")
 (3, "a")

julia> v = [(1, "c"), (3, "a"), (2, "b")]; sort!(v, by = x -> x[2]); v
3-element Vector{Tuple{Int64, String}}:
 (3, "a")
 (2, "b")
 (1, "c")

julia> sort(0:3, by=x->x-2, order=Base.Order.By(abs))
4-element Vector{Int64}:
 2
 1
 3
 0

julia> sort(0:3, by=x->x-2, order=Base.Order.By(abs)) == sort(0:3, by=x->abs(x-2))
true

julia> sort([2, NaN, 1, NaN, 3]) # correct sort with default lt=isless
5-element Vector{Float64}:
   1.0
   2.0
   3.0
 NaN
 NaN

julia> sort([2, NaN, 1, NaN, 3], lt=<) # wrong sort due to invalid lt. This behavior is
↪  undefined.
5-element Vector{Float64}:
   2.0
 NaN
   1.0
 NaN
   3.0
```

source

Base.sort – Function.

```
sort(A; dims::Integer, alg::Base.Sort.Algorithm=Base.Sort.defalg(A), lt=isless, by=identity,
↪  rev::Bool=false, order::Base.Order.Ordering=Base.Order.Forward)
```

Sort a multidimensional array A along the given dimension. See sort! for a description of possible keyword arguments.

To sort slices of an array, refer to sortslices.

Examples

```
julia> A = [4 3; 1 2]
2×2 Matrix{Int64}:
```

```
 4  3
 1  2

julia> sort(A, dims = 1)
2×2 Matrix{Int64}:
 1  2
 4  3

julia> sort(A, dims = 2)
2×2 Matrix{Int64}:
 3  4
 1  2
```

source

```
sort(v; alg::Base.Sort.Algorithm=Base.Sort.defalg(v), lt=isless, by=identity, rev::Bool=false,
↪  order::Base.Order.Ordering=Base.Order.Forward)
sort(v::NTuple; kws...)::NTuple
```

Variant of `sort!` that returns a sorted copy of v leaving v itself unmodified.

When calling `sort` on the `keys` or `values` of a dictionary, v is collected and then sorted.

Julia 1.12

Sorting NTuples requires Julia 1.12 or later.

Julia 1.13

Sorting keys sets and values iterators requires Julia 1.13 or later.

Examples

```
julia> v = [3, 1, 2];

julia> sort(v)
3-element Vector{Int64}:
 1
 2
 3

julia> v
3-element Vector{Int64}:
 3
 1
 2

julia> sort(values(Dict('a'=>2, 'b'=>1)))
2-element Vector{Int64}:
 1
 2
```

source

`Base.sortperm` – Function.

```
sortperm(A; alg::Base.Sort.Algorithm=Base.Sort.DEFAULT_UNSTABLE, lt=isless, by=identity,
↪  rev::Bool=false, order::Base.Order.Ordering=Base.Order.Forward, [dims::Integer])
```

Return a permutation vector or array `I` that puts `A[I]` in sorted order along the given dimension. If `A` has more than one dimension, then the `dims` keyword argument must be specified. The order is specified using the same keywords as `sort!`. The permutation is guaranteed to be stable even if the sorting algorithm is unstable: the indices of equal elements will appear in ascending order.

See also `sortperm!`, `partialsortperm`, `invperm`, `indexin`. To sort slices of an array, refer to `sortslices`.

Julia 1.9

The method accepting `dims` requires at least Julia 1.9.

Examples

```
julia> v = [3, 1, 2];

julia> p = sortperm(v)
3-element Vector{Int64}:
 2
 3
 1

julia> v[p]
3-element Vector{Int64}:
 1
 2
 3

julia> A = [8 7; 5 6]
2×2 Matrix{Int64}:
 8  7
 5  6

julia> sortperm(A, dims = 1)
2×2 Matrix{Int64}:
 2  4
 1  3

julia> sortperm(A, dims = 2)
2×2 Matrix{Int64}:
 3  1
 2  4
```

source

`Base.Sort.InsertionSort` – Constant.

```
InsertionSort
```

Use the insertion sort algorithm.

Insertion sort traverses the collection one element at a time, inserting each element into its correct, sorted position in the output vector.

Characteristics:

- *stable*: preserves the ordering of elements that compare equal

(e.g. "a" and "A" in a sort of letters that ignores case).

- *in-place* in memory.
- *quadratic performance* in the number of elements to be sorted:

it is well-suited to small collections but should not be used for large ones.

source

`Base.Sort.MergeSort` – Constant.

```
MergeSort
```

Indicate that a sorting function should use the merge sort algorithm. Merge sort divides the collection into subcollections and repeatedly merges them, sorting each subcollection at each step, until the entire collection has been recombined in sorted form.

Characteristics:

- *stable*: preserves the ordering of elements that compare equal (e.g. "a" and "A" in a sort of letters that ignores case).
- *not in-place* in memory.
- *divide-and-conquer* sort strategy.
- *good performance* for large collections but typically not quite as fast as `QuickSort`.

source

`Base.Sort.QuickSort` – Constant.

```
QuickSort
```

Indicate that a sorting function should use the quick sort algorithm, which is *not* stable.

Characteristics:

- *not stable*: does not preserve the ordering of elements that compare equal (e.g. "a" and "A" in a sort of letters that ignores case).
- *in-place* in memory.
- *divide-and-conquer*: sort strategy similar to `MergeSort`.
- *good performance* for large collections.

source

`Base.Sort.PartialQuickSort` – Type.

```
PartialQuickSort{T <: Union{Integer,OrdinalRange}}
```

Indicate that a sorting function should use the partial quick sort algorithm. `PartialQuickSort(k)` is like `QuickSort`, but is only required to find and sort the elements that would end up in `v[k]` were `v` fully sorted.

Characteristics:

- *not stable*: does not preserve the ordering of elements that compare equal (e.g. "a" and "A" in a sort of letters that ignores case).
- *in-place* in memory.
- *divide-and-conquer*: sort strategy similar to `MergeSort`.

Note that `PartialQuickSort(k)` does not necessarily sort the whole array. For example,

```
julia> x = rand(100);

julia> k = 50:100;

julia> s1 = sort(x; alg=QuickSort);

julia> s2 = sort(x; alg=PartialQuickSort(k));

julia> map(issorted, (s1, s2))
(true, false)

julia> map(x->issorted(x[k]), (s1, s2))
(true, true)

julia> s1[k] == s2[k]
true
```

source

`Base.Sort.sortperm!` – Function.

```
sortperm!(ix, A; alg::Base.Sort.Algorithm=Base.Sort.DEFAULT_UNSTABLE, lt=isless, by=identity,
↪  rev::Bool=false, order::Base.Order.Ordering=Base.Order.Forward, [dims::Integer])
```

Like `sortperm`, but accepts a preallocated index vector or array `ix` with the same axes as `A`. `ix` is initialized to contain the values `LinearIndices(A)`.

Warning

Behavior can be unexpected when any mutated argument shares memory with any other argument.

Julia 1.9

The method accepting `dims` requires at least Julia 1.9.

Examples

```
julia> v = [3, 1, 2]; p = zeros(Int, 3);

julia> sortperm!(p, v); p
3-element Vector{Int64}:
 2
 3
 1

julia> v[p]
3-element Vector{Int64}:
 1
 2
 3

julia> A = [8 7; 5 6]; p = zeros(Int,2, 2);

julia> sortperm!(p, A; dims=1); p
2×2 Matrix{Int64}:
 2  4
 1  3

julia> sortperm!(p, A; dims=2); p
2×2 Matrix{Int64}:
 3  1
 2  4
```

source

Base.sortslices – Function.

```
sortslices(A; dims, alg::Algorithm=DEFAULT_UNSTABLE, lt=isless, by=identity, rev::Bool=false,
↪  order::Ordering=Forward)
```

Sort slices of an array A. The required keyword argument dims must be either an integer or a tuple of integers. It specifies the dimension(s) over which the slices are sorted.

E.g., if A is a matrix, dims=1 will sort rows, dims=2 will sort columns. Note that the default comparison function on one dimensional slices sorts lexicographically.

For the remaining keyword arguments, see the documentation of sort!.

Examples

```
julia> sortslices([7 3 5; -1 6 4; 9 -2 8], dims=1) # Sort rows
3×3 Matrix{Int64}:
 -1   6  4
  7   3  5
  9  -2  8

julia> sortslices([7 3 5; -1 6 4; 9 -2 8], dims=1, lt=(x,y)->isless(x[2],y[2]))
3×3 Matrix{Int64}:
  9  -2  8
  7   3  5
 -1   6  4
```

```
julia> sortslices([7 3 5; -1 6 4; 9 -2 8], dims=1, rev=true)
3×3 Matrix{Int64}:
  9  -2  8
  7   3  5
 -1   6  4

julia> sortslices([7 3 5; 6 -1 -4; 9 -2 8], dims=2) # Sort columns
3×3 Matrix{Int64}:
  3   5  7
 -1  -4  6
 -2   8  9

julia> sortslices([7 3 5; 6 -1 -4; 9 -2 8], dims=2, alg=InsertionSort,
↪  lt=(x,y)->isless(x[2],y[2]))
3×3 Matrix{Int64}:
  5   3  7
 -4  -1  6
  8  -2  9

julia> sortslices([7 3 5; 6 -1 -4; 9 -2 8], dims=2, rev=true)
3×3 Matrix{Int64}:
 7   5   3
 6  -4  -1
 9   8  -2
```

Higher dimensions

`sortslices` extends naturally to higher dimensions. E.g., if `A` is a a 2x2x2 array, `sortslices(A, dims=3)` will sort slices within the 3rd dimension, passing the 2x2 slices `A[:, :, 1]` and `A[:, :, 2]` to the comparison function. Note that while there is no default order on higher-dimensional slices, you may use the by or `lt` keyword argument to specify such an order.

If `dims` is a tuple, the order of the dimensions in `dims` is relevant and specifies the linear order of the slices. E.g., if `A` is three dimensional and `dims` is `(1, 2)`, the orderings of the first two dimensions are re-arranged such that the slices (of the remaining third dimension) are sorted. If `dims` is `(2, 1)` instead, the same slices will be taken, but the result order will be row-major instead.

Higher dimensional examples

```
julia> A = [4 3; 2 1 ;;; 'A' 'B'; 'C' 'D']
2×2×2 Array{Any, 3}:
[:, :, 1] =
 4  3
 2  1

[:, :, 2] =
 'A'  'B'
 'C'  'D'

julia> sortslices(A, dims=(1,2))
2×2×2 Array{Any, 3}:
[:, :, 1] =
 1  3
 2  4
```

```
[:, :, 2] =
 'D'  'B'
 'C'  'A'

julia> sortslices(A, dims=(2,1))
2×2×2 Array{Any, 3}:
[:, :, 1] =
 1  2
 3  4

[:, :, 2] =
 'D'  'C'
 'B'  'A'

julia> sortslices(reshape([5; 4; 3; 2; 1], (1,1,5)), dims=3, by=x->x[1,1])
1×1×5 Array{Int64, 3}:
[:, :, 1] =
 1

[:, :, 2] =
 2

[:, :, 3] =
 3

[:, :, 4] =
 4

[:, :, 5] =
 5
```

source

57.2 Order-Related Functions

Base.issorted – Function.

```
issorted(itr, lt=isless, by=identity, rev::Bool=false,
↪  order::Base.Order.Ordering=Base.Order.Forward)
```

Test whether a collection is in sorted order. The keywords modify what order is considered sorted, as described in the sort! documentation.

Examples

```
julia> issorted([1, 2, 3])
true

julia> issorted([(1, "b"), (2, "a")], by = x -> x[1])
true

julia> issorted([(1, "b"), (2, "a")], by = x -> x[2])
```

```
false

julia> issorted([(1, "b"), (2, "a")], by = x -> x[2], rev=true)
true

julia> issorted([1, 2, -2, 3], by=abs)
true
```

source

Base.Sort.searchsorted – Function.

```
searchsorted(v, x; by=identity, lt=isless, rev=false)
```

Return the range of indices in v where values are equivalent to x, or an empty range located at the insertion point if v does not contain values equivalent to x. The vector v must be sorted according to the order defined by the keywords. Refer to sort! for the meaning of the keywords and the definition of equivalence. Note that the by function is applied to the searched value x as well as the values in v.

The range is generally found using binary search, but there are optimized implementations for some inputs.

See also: searchsortedfirst, sort!, insorted, findall.

Examples

```
julia> searchsorted([1, 2, 4, 5, 5, 7], 4) # single match
3:3

julia> searchsorted([1, 2, 4, 5, 5, 7], 5) # multiple matches
4:5

julia> searchsorted([1, 2, 4, 5, 5, 7], 3) # no match, insert in the middle
3:2 (empty range)

julia> searchsorted([1, 2, 4, 5, 5, 7], 9) # no match, insert at end
7:6 (empty range)

julia> searchsorted([1, 2, 4, 5, 5, 7], 0) # no match, insert at start
1:0 (empty range)

julia> searchsorted([1=>"one", 2=>"two", 2=>"two", 4=>"four"], 2=>"two", by=first) # compare
↪  the keys of the pairs
2:3
```

source

Base.Sort.searchsortedfirst – Function.

```
searchsortedfirst(v, x; by=identity, lt=isless, rev=false)
```

Return the index of the first value in v that is not ordered before x. If all values in v are ordered before x, return `lastindex(v) + 1`.

The vector v must be sorted according to the order defined by the keywords. `insert!`ing x at the returned index will maintain the sorted order. Refer to `sort!` for the meaning and use of the keywords. Note that the by function is applied to the searched value x as well as the values in v.

The index is generally found using binary search, but there are optimized implementations for some inputs.

See also: `searchsortedlast`, `searchsorted`, `findfirst`.

Examples

```
julia> searchsortedfirst([1, 2, 4, 5, 5, 7], 4) # single match
3

julia> searchsortedfirst([1, 2, 4, 5, 5, 7], 5) # multiple matches
4

julia> searchsortedfirst([1, 2, 4, 5, 5, 7], 3) # no match, insert in the middle
3

julia> searchsortedfirst([1, 2, 4, 5, 5, 7], 9) # no match, insert at end
7

julia> searchsortedfirst([1, 2, 4, 5, 5, 7], 0) # no match, insert at start
1

julia> searchsortedfirst([1=>"one", 2=>"two", 4=>"four"], 3=>"three", by=first) # compare the
↪  keys of the pairs
3
```

source

`Base.Sort.searchsortedlast` – Function.

```
searchsortedlast(v, x; by=identity, lt=isless, rev=false)
```

Return the index of the last value in v that is not ordered after x. If all values in v are ordered after x, return `firstindex(v) - 1`.

The vector v must be sorted according to the order defined by the keywords. `insert!`ing x immediately after the returned index will maintain the sorted order. Refer to `sort!` for the meaning and use of the keywords. Note that the by function is applied to the searched value x as well as the values in v.

The index is generally found using binary search, but there are optimized implementations for some inputs

Examples

```
julia> searchsortedlast([1, 2, 4, 5, 5, 7], 4) # single match
3

julia> searchsortedlast([1, 2, 4, 5, 5, 7], 5) # multiple matches
5

julia> searchsortedlast([1, 2, 4, 5, 5, 7], 3) # no match, insert in the middle
```

```
2

julia> searchsortedlast([1, 2, 4, 5, 5, 7], 9) # no match, insert at end
6

julia> searchsortedlast([1, 2, 4, 5, 5, 7], 0) # no match, insert at start
0

julia> searchsortedlast([1=>"one", 2=>"two", 4=>"four"], 3=>"three", by=first) # compare the
↪  keys of the pairs
2
```

source

Base.Sort.insorted – Function.

```
insorted(x, v; by=identity, lt=isless, rev=false)::Bool
```

Determine whether a vector v contains any value equivalent to x. The vector v must be sorted according to the order defined by the keywords. Refer to sort! for the meaning of the keywords and the definition of equivalence. Note that the by function is applied to the searched value x as well as the values in v.

The check is generally done using binary search, but there are optimized implementations for some inputs.

See also in.

Examples

```
julia> insorted(4, [1, 2, 4, 5, 5, 7]) # single match
true

julia> insorted(5, [1, 2, 4, 5, 5, 7]) # multiple matches
true

julia> insorted(3, [1, 2, 4, 5, 5, 7]) # no match
false

julia> insorted(9, [1, 2, 4, 5, 5, 7]) # no match
false

julia> insorted(0, [1, 2, 4, 5, 5, 7]) # no match
false

julia> insorted(2=>"TWO", [1=>"one", 2=>"two", 4=>"four"], by=first) # compare the keys of the
↪  pairs
true
```

Julia 1.6

insorted was added in Julia 1.6.

source

Base.Sort.partialsort! – Function.

```
partialsort!(v, k; by=identity, lt=isless, rev=false,
↪  order::Base.Order.Ordering=Base.Order.Forward)
```

Mutate the vector v so that the value at index k (or range of adjacent values if k is a range) occurs at the position where it would appear if the array were fully sorted. If k is a single index, that value is returned; if k is a range, an array of values at those indices is returned. Note that `partialsort!` may not fully sort the input array.

For the keyword arguments, see the documentation of `sort!`.

Examples

```
julia> a = [1, 2, 4, 3, 4]
5-element Vector{Int64}:
 1
 2
 4
 3
 4

julia> partialsort!(a, 4)
4

julia> a
5-element Vector{Int64}:
 1
 2
 3
 4
 4

julia> a = [1, 2, 4, 3, 4]
5-element Vector{Int64}:
 1
 2
 4
 3
 4

julia> partialsort!(a, 4, rev=true)
2

julia> a
5-element Vector{Int64}:
 4
 4
 3
 2
 1
```

source

Base.Sort.partialsort – Function.

```
partialsort(v, k, by=identity, lt=isless, rev=false,
↪  order::Base.Order.Ordering=Base.Order.Forward)
```

Variant of `partialsort!` that copies v before partially sorting it, thereby returning the same thing as `partialsort!` but leaving v unmodified.

source

`Base.Sort.partialsortperm` – Function.

```
partialsortperm(v, k; by=identity, lt=isless, rev=false)
```

Return a partial permutation `I` of the vector v, so that `v[I]` returns values of a fully sorted version of v at index k. If k is a range, a vector of indices is returned; if k is an integer, a single index is returned. The order is specified using the same keywords as `sort!`. The permutation is stable: the indices of equal elements will appear in ascending order.

This function is equivalent to, but more efficient than, calling `sortperm(...)[k]`.

Examples

```
julia> v = [3, 1, 2, 1];

julia> v[partialsortperm(v, 1)]
1

julia> p = partialsortperm(v, 1:3)
3-element view(::Vector{Int64}, 1:3) with eltype Int64:
 2
 4
 3

julia> v[p]
3-element Vector{Int64}:
 1
 1
 2
```

source

`Base.Sort.partialsortperm!` – Function.

```
partialsortperm!(ix, v, k; by=identity, lt=isless, rev=false)
```

Like `partialsortperm`, but accepts a preallocated index vector `ix` the same size as v, which is used to store (a permutation of) the indices of v.

`ix` is initialized to contain the indices of v.

(Typically, the indices of v will be `1:length(v)`, although if v has an alternative array type with non-one-based indices, such as an `OffsetArray`, `ix` must share those same indices)

Upon return, `ix` is guaranteed to have the indices k in their sorted positions, such that

```
partialsortperm!(ix, v, k);
v[ix[k]] == partialsort(v, k)
```

The return value is the kth element of `ix` if k is an integer, or view into `ix` if k is a range.

Warning

Behavior can be unexpected when any mutated argument shares memory with any other argument.

Examples

```
julia> v = [3, 1, 2, 1];

julia> ix = Vector{Int}(undef, 4);

julia> partialsortperm!(ix, v, 1)
2

julia> ix = [1:4;];

julia> partialsortperm!(ix, v, 2:3)
2-element view(::Vector{Int64}, 2:3) with eltype Int64:
 4
 3
```

source

57.3 Sorting Algorithms

There are currently four sorting algorithms publicly available in base Julia:

- `InsertionSort`
- `QuickSort`
- `PartialQuickSort(k)`
- `MergeSort`

By default, the `sort` family of functions uses stable sorting algorithms that are fast on most inputs. The exact algorithm choice is an implementation detail to allow for future performance improvements. Currently, a hybrid of `RadixSort`, `ScratchQuickSort`, `InsertionSort`, and `CountingSort` is used based on input type, size, and composition. Implementation details are subject to change but currently available in the extended help of `??Base.DEFAULT_STABLE` and the docstrings of internal sorting algorithms listed there.

You can explicitly specify your preferred algorithm with the `alg` keyword (e.g. `sort!(v, alg=PartialQuickSort(10:20))`) or reconfigure the default sorting algorithm for custom types by adding a specialized method to the `Base.Sort.defalg` function. For example, InlineStrings.jl defines the following method:

```
Base.Sort.defalg(::AbstractArray{<:Union{SmallInlineStrings, Missing}}) = InlineStringSort
```

Julia 1.9

The default sorting algorithm (returned by `Base.Sort.defalg`) is guaranteed to be stable since Julia 1.9. Previous versions had unstable edge cases when sorting numeric arrays.

57.4 Alternate Orderings

By default, `sort`, `searchsorted`, and related functions use `isless` to compare two elements in order to determine which should come first. The `Base.Order.Ordering` abstract type provides a mechanism for defining alternate orderings on the same set of elements: when calling a sorting function like `sort!`, an instance of `Ordering` can be provided with the keyword argument `order`.

Instances of `Ordering` define an order through the `Base.Order.lt` function, which works as a generalization of `isless`. This function's behavior on custom `Orderings` must satisfy all the conditions of a strict weak order. See `sort!` for details and examples of valid and invalid `lt` functions.

`Base.Order.Ordering` – Type.

```
Base.Order.Ordering
```

Abstract type which represents a strict weak order on some set of elements. See `sort!` for more.

Use `Base.Order.lt` to compare two elements according to the ordering.

source

`Base.Order.lt` – Function.

```
lt(o::Ordering, a, b)::Bool
```

Test whether a is less than b according to the ordering `o`.

source

`Base.Order.ord` – Function.

```
ord(lt, by, rev::Union{Bool, Nothing}, order::Ordering=Forward)
```

Construct an `Ordering` object from the same arguments used by `sort!`. Elements are first transformed by the function by (which may be `identity`) and are then compared according to either the function `lt` or an existing ordering `order`. `lt` should be `isless` or a function that obeys the same rules as the `lt` parameter of `sort!`. Finally, the resulting order is reversed if `rev=true`.

Passing an `lt` other than `isless` along with an `order` other than `Base.Order.Forward` or `Base.Order.Reverse` is not permitted, otherwise all options are independent and can be used together in all possible combinations.

source

`Base.Order.Forward` – Constant.

```
Base.Order.Forward
```

Default ordering according to `isless`.

source

Base.Order.ReverseOrdering – Type.

```
ReverseOrdering(fwd::Ordering=Forward)
```

A wrapper which reverses an ordering.

For a given `Ordering` `o`, the following holds for all `a`, `b`:

```
lt(ReverseOrdering(o), a, b) == lt(o, b, a)
```

source

Base.Order.Reverse – Constant.

```
Base.Order.Reverse
```

Reverse ordering according to `isless`.

source

Base.Order.By – Type.

```
By(by, order::Ordering=Forward)
```

`Ordering` which applies `order` to elements after they have been transformed by the function `by`.

source

Base.Order.Lt – Type.

```
Lt(lt)
```

`Ordering` that calls `lt(a, b)` to compare elements. `lt` must obey the same rules as the `lt` parameter of `sort!`.

source

Base.Order.Perm – Type.

```
Perm(order::Ordering, data::AbstractVector)
```

`Ordering` on the indices of `data` where `i` is less than `j` if `data[i]` is less than `data[j]` according to `order`. In the case that `data[i]` and `data[j]` are equal, `i` and `j` are compared by numeric value.

source

Chapter 58

Iteration utilities

`Base.Iterators.Stateful` – Type.

```
Stateful(itr)
```

There are several different ways to think about this iterator wrapper:

1. It provides a mutable wrapper around an iterator and its iteration state.
2. It turns an iterator-like abstraction into a `Channel`-like abstraction.
3. It's an iterator that mutates to become its own rest iterator whenever an item is produced.

`Stateful` provides the regular iterator interface. Like other mutable iterators (e.g. `Base.Channel`), if iteration is stopped early (e.g. by a `break` in a `for` loop), iteration can be resumed from the same spot by continuing to iterate over the same iterator object (in contrast, an immutable iterator would restart from the beginning).

Examples

```
julia> a = Iterators.Stateful("abcdef");

julia> isempty(a)
false

julia> popfirst!(a)
'a': ASCII/Unicode U+0061 (category Ll: Letter, lowercase)

julia> collect(Iterators.take(a, 3))
3-element Vector{Char}:
 'b': ASCII/Unicode U+0062 (category Ll: Letter, lowercase)
 'c': ASCII/Unicode U+0063 (category Ll: Letter, lowercase)
 'd': ASCII/Unicode U+0064 (category Ll: Letter, lowercase)

julia> collect(a)
2-element Vector{Char}:
 'e': ASCII/Unicode U+0065 (category Ll: Letter, lowercase)
 'f': ASCII/Unicode U+0066 (category Ll: Letter, lowercase)
```

```
julia> Iterators.reset!(a); popfirst!(a)
'a': ASCII/Unicode U+0061 (category Ll: Letter, lowercase)

julia> Iterators.reset!(a, "hello"); popfirst!(a)
'h': ASCII/Unicode U+0068 (category Ll: Letter, lowercase)
```

```
julia> a = Iterators.Stateful([1,1,1,2,3,4]);

julia> for x in a; x == 1 || break; end

julia> peek(a)
3

julia> sum(a) # Sum the remaining elements
7
```

source

`Base.Iterators.zip` – Function.

```
zip(iters...)
```

Run multiple iterators at the same time, until any of them is exhausted. The value type of the `zip` iterator is a tuple of values of its subiterators.

Note

`zip` orders the calls to its subiterators in such a way that stateful iterators will not advance when another iterator finishes in the current iteration.

Note

`zip()` with no arguments yields an infinite iterator of empty tuples.

See also: `enumerate`, `Base.splat`.

Examples

```
julia> a = 1:5
1:5

julia> b = ["e","d","b","c","a"]
5-element Vector{String}:
 "e"
 "d"
 "b"
 "c"
 "a"

julia> c = zip(a,b)
zip(1:5, ["e", "d", "b", "c", "a"])

julia> length(c)
5
```

```
julia> first(c)
(1, "e")
```

source

Base.Iterators.enumerate – Function.

```
enumerate(iter)
```

An iterator that yields `(i, x)` where `i` is a counter starting at 1, and `x` is the `i`th value from the given iterator. It's useful when you need not only the values `x` over which you are iterating, but also the number of iterations so far.

Note that `i` may not be valid for indexing `iter`, or may index a different element. This will happen if `iter` has indices that do not start at 1, and may happen for strings, dictionaries, etc. See the `pairs(IndexLinear(), iter)` method if you want to ensure that `i` is an index.

Examples

```
julia> a = ["a", "b", "c"];

julia> for (index, value) in enumerate(a)
           println("$index $value")
       end
1 a
2 b
3 c

julia> str = "naïve";

julia> for (i, val) in enumerate(str)
           print("i = ", i, ", val = ", val, ", ")
           try @show(str[i]) catch e println(e) end
       end
i = 1, val = n, str[i] = 'n'
i = 2, val = a, str[i] = 'a'
i = 3, val = ï, str[i] = 'ï'
i = 4, val = v, StringIndexError("naïve", 4)
i = 5, val = e, str[i] = 'v'
```

source

Base.Iterators.rest – Function.

```
rest(iter, state)
```

An iterator that yields the same elements as `iter`, but starting at the given `state`, which must be a state obtainable via a sequence of one or more calls to `iterate(iter[, state])`

See also: `Iterators.drop`, `Iterators.peel`, `Base.rest`.

Examples

```
julia> iter = [1,2,3,4];

julia> val, state = iterate(iter)
(1, 2)

julia> collect(Iterators.rest(iter, state))
3-element Vector{Int64}:
 2
 3
 4
```

source

Base.Iterators.countfrom – Function.

```
countfrom(start=1, step=1)
```

An iterator that counts forever, starting at `start` and incrementing by `step`.

Examples

```
julia> for v in Iterators.countfrom(5, 2)
           v > 10 && break
           println(v)
       end
5
7
9
```

source

Base.Iterators.take – Function.

```
take(iter, n)
```

An iterator that generates at most the first n elements of `iter`.

See also: `drop`, `peel`, `first`, `Base.take!`.

Examples

```
julia> a = 1:2:11
1:2:11

julia> collect(a)
6-element Vector{Int64}:
  1
  3
  5
  7
  9
 11

julia> collect(Iterators.take(a,3))
3-element Vector{Int64}:
```

```
1
3
5
```

source

Base.Iterators.takewhile – Function.

```
takewhile(pred, iter)
```

An iterator that generates element from `iter` as long as predicate `pred` is true, afterwards, drops every element.

Julia 1.4

This function requires at least Julia 1.4.

Examples

```
julia> s = collect(1:5)
5-element Vector{Int64}:
 1
 2
 3
 4
 5

julia> collect(Iterators.takewhile(<(3),s))
2-element Vector{Int64}:
 1
 2
```

source

Base.Iterators.drop – Function.

```
drop(iter, n)
```

An iterator that generates all but the first `n` elements of `iter`.

Examples

```
julia> a = 1:2:11
1:2:11

julia> collect(a)
6-element Vector{Int64}:
  1
  3
  5
  7
  9
 11
```

```
julia> collect(Iterators.drop(a,4))
2-element Vector{Int64}:
  9
 11
```

source

Base.Iterators.dropwhile – Function.

```
dropwhile(pred, iter)
```

An iterator that drops element from `iter` as long as predicate `pred` is true, afterwards, returns every element.

Julia 1.4

This function requires at least Julia 1.4.

Examples

```
julia> s = collect(1:5)
5-element Vector{Int64}:
 1
 2
 3
 4
 5

julia> collect(Iterators.dropwhile(<(3),s))
3-element Vector{Int64}:
 3
 4
 5
```

source

Base.Iterators.findeach – Function.

```
findeach(f, it)
findeach(it)
```

An iterator that generates every key from the key/value pairs of `pairs(it)`, where `f(value)` returns `true`.

If `f` is not specified, default to `identity`.

`Iterators.findeach` is the lazy equivalent of `findall`.

Julia 1.13

`findeach` requires at least Julia 1.13.

Examples

```
julia> collect(Iterators.findeach(isodd, Dict(2 => 3, 3 => 2)))
1-element Vector{Int64}:
 2

julia> only(Iterators.findeach(==(1), [3,6,2,1]))
4
```

source

Base.Iterators.nth – Function.

```
nth(n::Integer)
```

Return a function that gets the n-th element from any iterator passed to it. Equivalent to `Base.Fix2(nth, n)` or `itr -> nth(itr, n)`.

See also: nth, Base.Fix2

Examples

```
julia> fifth_element = Iterators.nth(5)
(::Base.Fix2{typeof(Base.Iterators.nth), Int64}) (generic function with 2 methods)

julia> fifth_element(reshape(1:30, (5,6)))
5

julia> map(fifth_element, ("Willis", "Jovovich", "Oldman"))
('i', 'v', 'a')
```

source

```
nth(itr, n::Integer)
```

Get the nth element of an iterable collection. Throw a `BoundsError`[@ref] if not existing. Will advance any `Stateful`[@ref] iterator.

See also: first, last

Examples

```
julia> Iterators.nth(2:2:10, 4)
8

julia> Iterators.nth(reshape(1:30, (5,6)), 6)
6

julia> stateful = Iterators.Stateful(1:10); Iterators.nth(stateful, 7)
7

julia> first(stateful)
8
```

Julia 1.13

This function requires at least Julia 1.13.

source

Base.Iterators.cycle – Function.

```
cycle(iter[, n::Int])
```

An iterator that cycles through `iter` forever. If `n` is specified, then it cycles through `iter` that many times. When `iter` is empty, so are `cycle(iter)` and `cycle(iter, n)`.

`Iterators.cycle(iter, n)` is the lazy equivalent of `Base.repeat(vector, n)`, while `Iterators.repeated(iter, n)` is the lazy `Base.fill(item, n)`.

Julia 1.11

The method `cycle(iter, n)` was added in Julia 1.11.

Examples

```
julia> for (i, v) in enumerate(Iterators.cycle("hello"))
           print(v)
           i > 10 && break
       end
hellohelloh

julia> foreach(print, Iterators.cycle(['j', 'u', 'l', 'i', 'a'], 3))
juliajuliajulia

julia> repeat([1,2,3], 4) == collect(Iterators.cycle([1,2,3], 4))
true

julia> fill([1,2,3], 4) == collect(Iterators.repeated([1,2,3], 4))
true
```

source

Base.Iterators.repeated – Function.

```
repeated(x[, n::Int])
```

An iterator that generates the value `x` forever. If `n` is specified, generates `x` that many times (equivalent to `take(repeated(x), n)`).

See also `fill`, and compare `Iterators.cycle`.

Examples

```
julia> a = Iterators.repeated([1 2], 4);

julia> collect(a)
4-element Vector{Matrix{Int64}}:
 [1 2]
 [1 2]
 [1 2]
 [1 2]
```

```
julia> ans == fill([1 2], 4)
true

julia> Iterators.cycle([1 2], 4) |> collect |> println
[1, 2, 1, 2, 1, 2, 1, 2]
```

source

Base.Iterators.product – Function.

```
product(iters...)
```

Return an iterator over the product of several iterators. Each generated element is a tuple whose `i`th element comes from the `i`th argument iterator. The first iterator changes the fastest.

See also: `zip`, `Iterators.flatten`.

Examples

```
julia> collect(Iterators.product(1:2, 3:5))
2×3 Matrix{Tuple{Int64, Int64}}:
 (1, 3)  (1, 4)  (1, 5)
 (2, 3)  (2, 4)  (2, 5)

julia> ans == [(x,y) for x in 1:2, y in 3:5]  # collects a generator involving
↪  Iterators.product
true
```

source

Base.Iterators.flatten – Function.

```
flatten(iter)
```

Given an iterator that yields iterators, return an iterator that yields the elements of those iterators. Put differently, the elements of the argument iterator are concatenated.

Examples

```
julia> collect(Iterators.flatten((1:2, 8:9)))
4-element Vector{Int64}:
 1
 2
 8
 9

julia> [(x,y) for x in 0:1 for y in 'a':'c']  # collects generators involving
↪  Iterators.flatten
6-element Vector{Tuple{Int64, Char}}:
 (0, 'a')
 (0, 'b')
 (0, 'c')
 (1, 'a')
 (1, 'b')
 (1, 'c')
```

source

Base.Iterators.flatmap – Function.

```
Iterators.flatmap(f, iterators...)
```

Equivalent to `flatten(map(f, iterators...))`.

See also `Iterators.flatten`, `Iterators.map`.

Julia 1.9

This function was added in Julia 1.9.

Examples

```
julia> Iterators.flatmap(n -> -n:2:n, 1:3) |> collect
9-element Vector{Int64}:
 -1
  1
 -2
  0
  2
 -3
 -1
  1
  3

julia> stack(n -> -n:2:n, 1:3)
ERROR: DimensionMismatch: stack expects uniform slices, got axes(x) == (1:3,) while first had
↪  (1:2,)
[...]

julia> Iterators.flatmap(n -> (-n, 10n), 1:2) |> collect
4-element Vector{Int64}:
 -1
 10
 -2
 20

julia> ans == vec(stack(n -> (-n, 10n), 1:2))
true
```

source

Base.Iterators.partition – Function.

```
partition(collection, n)
```

Iterate over a collection n elements at a time.

Examples

```
julia> collect(Iterators.partition([1,2,3,4,5], 2))
3-element Vector{SubArray{Int64, 1, Vector{Int64}, Tuple{UnitRange{Int64}}, true}}:
```

```
[1, 2]
[3, 4]
[5]
```

source

Base.Iterators.map – Function.

```
Iterators.map(f, iterators...)
```

Create a lazy mapping. This is another syntax for writing `(f(args...) for args in zip(iterators...))`.

Julia 1.6

This function requires at least Julia 1.6.

Examples

```
julia> collect(Iterators.map(x -> x^2, 1:3))
3-element Vector{Int64}:
 1
 4
 9
```

source

Base.Iterators.filter – Function.

```
Iterators.filter(flt, itr)
```

Given a predicate function `flt` and an iterable object `itr`, return an iterable object which upon iteration yields the elements `x` of `itr` that satisfy `flt(x)`. The order of the original iterator is preserved.

This function is *lazy*; that is, it is guaranteed to return in (1) time and use (1) additional space, and `flt` will not be called by an invocation of `filter`. Calls to `flt` will be made when iterating over the returned iterable object. These calls are not cached and repeated calls will be made when reiterating.

Warning

Subsequent *lazy* transformations on the iterator returned from `filter`, such as those performed by `Iterators.reverse` or `cycle`, will also delay calls to `flt` until collecting or iterating over the returned iterable object. If the filter predicate is nondeterministic or its return values depend on the order of iteration over the elements of `itr`, composition with lazy transformations may result in surprising behavior. If this is undesirable, either ensure that `flt` is a pure function or collect intermediate `filter` iterators before further transformations.

See `Base.filter` for an eager implementation of filtering for arrays.

Examples

```
julia> f = Iterators.filter(isodd, [1, 2, 3, 4, 5])
Base.Iterators.Filter{typeof(isodd), Vector{Int64}}(isodd, [1, 2, 3, 4, 5])

julia> foreach(println, f)
```

```
1
3
5

julia> [x for x in [1, 2, 3, 4, 5] if isodd(x)]  # collects a generator over Iterators.filter
3-element Vector{Int64}:
 1
 3
 5
```

source

Base.Iterators.accumulate – Function.

```
Iterators.accumulate(f, itr; [init])
```

Given a 2-argument function `f` and an iterator `itr`, return a new iterator that successively applies `f` to the previous value and the next element of `itr`.

This is effectively a lazy version of `Base.accumulate`.

Julia 1.5

Keyword argument `init` is added in Julia 1.5.

Examples

```
julia> a = Iterators.accumulate(+, [1,2,3,4]);

julia> foreach(println, a)
1
3
6
10

julia> b = Iterators.accumulate(/, (2, 5, 2, 5); init = 100);

julia> collect(b)
4-element Vector{Float64}:
 50.0
 10.0
  5.0
  1.0
```

source

Base.Iterators.reverse – Function.

```
Iterators.reverse(itr)
```

Given an iterator `itr`, then `reverse(itr)` is an iterator over the same collection but in the reverse order. This iterator is "lazy" in that it does not make a copy of the collection in order to reverse it; see `Base.reverse` for an eager implementation.

(By default, this returns an `Iterators.Reverse` object wrapping `itr`, which is iterable if the corresponding `iterate` methods are defined, but some `itr` types may implement more specialized `Iterators.reverse` behaviors.)

Not all iterator types T support reverse-order iteration. If T doesn't, then iterating over `Iterators.reverse(itr::T)` will throw a `MethodError` because of the missing `iterate` methods for `Iterators.Reverse{T}`. (To implement these methods, the original iterator `itr::T` can be obtained from an `r::Iterators.Reverse{T}` object by `r.itr`; more generally, one can use `Iterators.reverse(r)`.)

Examples

```
julia> foreach(println, Iterators.reverse(1:5))
5
4
3
2
1
```

source

Base.Iterators.only – Function.

```
only(x)
```

Return the one and only element of collection `x`, or throw an `ArgumentError` if the collection has zero or multiple elements.

See also `first`, `last`.

Julia 1.4

This method requires at least Julia 1.4.

Examples

```
julia> only(["a"])
"a"

julia> only("a")
'a': ASCII/Unicode U+0061 (category Ll: Letter, lowercase)

julia> only(())
ERROR: ArgumentError: Tuple contains 0 elements, must contain exactly 1 element
Stacktrace:
[...]

julia> only(('a', 'b'))
ERROR: ArgumentError: Tuple contains 2 elements, must contain exactly 1 element
Stacktrace:
[...]
```

source

Base.Iterators.peel – Function.

```
peel(iter)
```

Returns the first element and an iterator over the remaining elements.

If the iterator is empty return `nothing` (like `iterate`).

Julia 1.7

Prior versions throw a BoundsError if the iterator is empty.

See also: `Iterators.drop`, `Iterators.take`.

Examples

```
julia> (a, rest) = Iterators.peel("abc");

julia> a
'a': ASCII/Unicode U+0061 (category Ll: Letter, lowercase)

julia> collect(rest)
2-element Vector{Char}:
 'b': ASCII/Unicode U+0062 (category Ll: Letter, lowercase)
 'c': ASCII/Unicode U+0063 (category Ll: Letter, lowercase)
```

source

Chapter 59

Reflection and introspection

Julia provides a variety of runtime reflection capabilities.

59.1 Module bindings

The public names for a `Module` are available using `names(m::Module)`, which will return an array of `Symbol` elements representing the public bindings. `names(m::Module, all = true)` returns symbols for all bindings in `m`, regardless of public status.

59.2 DataType fields

The names of `DataType` fields may be interrogated using `fieldnames`. For example, given the following type, `fieldnames(Point)` returns a tuple of `Symbol`s representing the field names:

```
julia> struct Point
           x::Int
           y
       end

julia> fieldnames(Point)
(:x, :y)
```

The type of each field in a `Point` object is stored in the `types` field of the `Point` variable itself:

```
julia> Point.types
svec(Int64, Any)
```

While `x` is annotated as an `Int`, `y` was unannotated in the type definition, therefore `y` defaults to the `Any` type.

Types are themselves represented as a structure called `DataType`:

```
julia> typeof(Point)
DataType
```

Note that `fieldnames(DataType)` gives the names for each field of `DataType` itself, and one of these fields is the `types` field observed in the example above.

59.3 Subtypes

The *direct* subtypes of any DataType may be listed using `subtypes`. For example, the abstract DataType `AbstractFloat` has four (concrete) subtypes:

```
julia> InteractiveUtils.subtypes(AbstractFloat)
5-element Vector{Any}:
 BigFloat
 Core.BFloat16
 Float16
 Float32
 Float64
```

Any abstract subtype will also be included in this list, but further subtypes thereof will not; recursive application of `subtypes` may be used to inspect the full type tree.

Note that `subtypes` is located inside `InteractiveUtils` but is automatically exported when using the REPL.

59.4 DataType layout

The internal representation of a DataType is critically important when interfacing with C code and several functions are available to inspect these details. `isbitstype(T::DataType)` returns true if T is stored with C-compatible alignment. `fieldoffset(T::DataType, i::Integer)` returns the (byte) offset for field *i* relative to the start of the type.

59.5 Function methods

The methods of any generic function may be listed using `methods`. The method dispatch table may be searched for methods accepting a given type using `methodswith`.

59.6 Expansion and lowering

As discussed in the Metaprogramming section, the `macroexpand` function gives the unquoted and interpolated expression (`Expr`) form for a given macro. To use `macroexpand`, `quote` the expression block itself (otherwise, the macro will be evaluated and the result will be passed instead!). For example:

```
julia> macroexpand(@__MODULE__, :(@invoke identity(1::Int)))
:(Core.invoke(identity, Base.Tuple{Int}, 1))
```

The functions `Base.Meta.show_sexpr` and `dump` are used to display S-expr style views and depth-nested detail views for any expression.

Finally, the `Meta.lower` function gives the `lowered` form of any expression and is of particular interest for understanding how language constructs map to primitive operations such as assignments, branches, and calls:

```
julia> Meta.lower(@__MODULE__, :( [1+2, sin(0.5)] ))
:($(Expr(:thunk, CodeInfo(
1 − %1 = :+
```

```
│   %2 =   dynamic (%1)(1, 2)
│   %3 = sin
│   %4 =   dynamic (%3)(0.5)
│   %5 =   dynamic Base.vect(%2, %4)
└──      return %5
))))
```

59.7 Intermediate and compiled representations

Inspecting the lowered form for functions requires selection of the specific method to display, because generic functions may have many methods with different type signatures. For this purpose, method-specific code-lowering is available using `code_lowered`, and the type-inferred form is available using `code_typed`. `code_warntype` adds highlighting to the output of `code_typed`.

Closer to the machine, the LLVM intermediate representation of a function may be printed using by `code_llvm`, and finally the compiled machine code is available using `code_native` (this will trigger JIT compilation/code generation for any function which has not previously been called).

For convenience, there are macro versions of the above functions which take standard function calls and expand argument types automatically:

```
julia> @code_llvm +(1,1)
;  @ int.jl:87 within `+`
; Function Attrs: sspstrong uwtable
define i64 @"julia_+_476"(i64 signext %0, i64 signext %1) #0 {
top:
  %2 = add i64 %1, %0
  ret i64 %2
}
```

For more information see `@code_lowered`, `@code_typed`, `@code_warntype`, `@code_llvm`, and `@code_native`.

Printing of debug information

The aforementioned functions and macros take the keyword argument `debuginfo` that controls the level of debug information printed.

```
julia> InteractiveUtils.@code_typed debuginfo=:source +(1,1)
CodeInfo(
    @ int.jl:87 within `+`
1 ─ %1 = intrinsic Base.add_int(x, y)::Int64
└──      return %1
) => Int64
```

Possible values for `debuginfo` are: `:none`, `:source`, and `:default`. Per default debug information is not printed, but that can be changed by setting `Base.IRShow.default_debuginfo[] = :source`.

Chapter 60

C Interface

`Base.@ccall` – Macro.

```
@ccall library.function_name(argvalue1::argtype1, ...)::returntype
@ccall function_name(argvalue1::argtype1, ...)::returntype
@ccall $function_pointer(argvalue1::argtype1, ...)::returntype
```

Call a function in a C-exported shared library, specified by `library.function_name`, where `library` is a string constant or literal. The library may be omitted, in which case the `function_name` is resolved in the current process. Alternatively, `@ccall` may also be used to call a function pointer `$function_pointer`, such as one returned by `dlsym`.

Each `argvalue` to `@ccall` is converted to the corresponding `argtype`, by automatic insertion of calls to `unsafe_convert(argtype, cconvert(argtype, argvalue))`. (See also the documentation for `unsafe_convert` and `cconvert` for further details.) In most cases, this simply results in a call to `convert(argtype, argvalue)`.

Examples

```
@ccall strlen(s::Cstring)::Csize_t
```

This calls the C standard library function:

```
size_t strlen(char *)
```

with a Julia variable named `s`. See also `ccall`.

Varargs are supported with the following convention:

```
@ccall printf("%s = %d"::Cstring ; "foo"::Cstring, foo::Cint)::Cint
```

The semicolon is used to separate required arguments (of which there must be at least one) from variadic arguments.

Example using an external library:

```
# C signature of g_uri_escape_string:
# char *g_uri_escape_string(const char *unescaped, const char *reserved_chars_allowed,
↪  gboolean allow_utf8);
```

```
const glib = "libglib-2.0"
@ccall glib.g_uri_escape_string(my_uri::Cstring, ":/"::Cstring, true::Cint)::Cstring
```

The string literal could also be used directly before the function name, if desired `"libglib-2.0".g_uri_escape_string(...`

It's possible to declare the ccall as `gc_safe` by using the `gc_safe = true` option:

```
@ccall gc_safe=true strlen(s::Cstring)::Csize_t
```

This allows the garbage collector to run concurrently with the ccall, which can be useful whenever the `ccall` may block outside of julia.

Warning

This option should be used with caution, as it can lead to undefined behavior if the ccall calls back into the julia runtime. (`@cfunction`/`@ccallables` are safe however)

Julia 1.12

The `gc_safe` argument requires Julia 1.12 or higher.

source

`ccall` – Keyword.

```
ccall((function_name, library), returntype, (argtype1, ...), argvalue1, ...)
ccall(function_name, returntype, (argtype1, ...), argvalue1, ...)
ccall(function_pointer, returntype, (argtype1, ...), argvalue1, ...)
```

Call a function in a C-exported shared library, specified by the tuple `(function_name, library)`, where each component is either a string or symbol. Instead of specifying a library, one can also use a `function_name` symbol or string, which is resolved in the current process. Alternatively, `ccall` may also be used to call a function pointer `function_pointer`, such as one returned by `dlsym`.

Note that the argument type tuple must be a literal tuple, and not a tuple-valued variable or expression.

Each `argvalue` to the `ccall` will be converted to the corresponding `argtype`, by automatic insertion of calls to `unsafe_convert(argtype, cconvert(argtype, argvalue))`. (See also the documentation for `unsafe_convert` and `cconvert` for further details.) In most cases, this simply results in a call to `convert(argtype, argvalue)`.

source

`Core.Intrinsics.cglobal` – Function.

```
cglobal((symbol, library) [, type=Cvoid])
```

Obtain a pointer to a global variable in a C-exported shared library, specified exactly as in `ccall`. Returns a `Ptr{Type}`, defaulting to `Ptr{Cvoid}` if no Type argument is supplied. The values can be read or written by `unsafe_load` or `unsafe_store!`, respectively.

source

`Base.@cfunction` – Macro.

```
@cfunction(callable, ReturnType, (ArgumentTypes...,)) -> Ptr{Cvoid}
@cfunction($callable, ReturnType, (ArgumentTypes...,)) -> CFunction
```

Generate a C-callable function pointer from the Julia function `callable` for the given type signature. To pass the return value to a `ccall`, use the argument type `Ptr{Cvoid}` in the signature.

Note that the argument type tuple must be a literal tuple, and not a tuple-valued variable or expression (although it can include a splat expression). And that these arguments will be evaluated in global scope during compile-time (not deferred until runtime). Adding a '$' in front of the function argument changes this to instead create a runtime closure over the local variable `callable` (this is not supported on all architectures).

See manual section on ccall and cfunction usage.

Examples

```
julia> function foo(x::Int, y::Int)
           return x + y
       end

julia> @cfunction(foo, Int, (Int, Int))
Ptr{Cvoid} @0x000000001b82fcd0
```

source

`Base.CFunction` – Type.

```
CFunction struct
```

Garbage-collection handle for the return value from `@cfunction` when the first argument is annotated with '$'. Like all `cfunction` handles, it should be passed to `ccall` as a `Ptr{Cvoid}`, and will be converted automatically at the call site to the appropriate type.

See `@cfunction`.

source

`Base.unsafe_convert` – Function.

```
unsafe_convert(T, x)
```

Convert x to a C argument of type T where the input x must be the return value of `cconvert(T, ...)`.

In cases where `convert` would need to take a Julia object and turn it into a `Ptr`, this function should be used to define and perform that conversion.

Be careful to ensure that a Julia reference to x exists as long as the result of this function will be used. Accordingly, the argument x to this function should never be an expression, only a variable name or field reference. For example, `x=a.b.c` is acceptable, but `x=[a,b,c]` is not.

The `unsafe` prefix on this function indicates that using the result of this function after the x argument to this function is no longer accessible to the program may cause undefined behavior, including program corruption or segfaults, at any later time.

See also `cconvert`

source

`Base.cconvert` – Function.

```
cconvert(T,x)
```

Convert x to a value to be passed to C code as type T, typically by calling `convert(T, x)`.

In cases where x cannot be safely converted to T, unlike `convert`, `cconvert` may return an object of a type different from T, which however is suitable for `unsafe_convert` to handle. The result of this function should be kept valid (for the GC) until the result of `unsafe_convert` is not needed anymore. This can be used to allocate memory that will be accessed by the `ccall`. If multiple objects need to be allocated, a tuple of the objects can be used as return value.

Neither `convert` nor `cconvert` should take a Julia object and turn it into a `Ptr`.

source

`Base.unsafe_load` – Function.

```
unsafe_load(p::Ptr{T}, i::Integer=1)
unsafe_load(p::Ptr{T}, order::Symbol)
unsafe_load(p::Ptr{T}, i::Integer, order::Symbol)
```

Load a value of type T from the address of the `i`th element (1-indexed) starting at `p`. This is equivalent to the C expression `p[i-1]`. Optionally, an atomic memory ordering can be provided.

The `unsafe` prefix on this function indicates that no validation is performed on the pointer `p` to ensure that it is valid. Like C, the programmer is responsible for ensuring that referenced memory is not freed or garbage collected while invoking this function. Incorrect usage may segfault your program or return garbage answers. Unlike C, dereferencing memory region allocated as different type may be valid provided that the types are compatible.

Julia 1.10

The `order` argument is available as of Julia 1.10.

See also: `atomic`

source

`Base.unsafe_store!` – Function.

```
unsafe_store!(p::Ptr{T}, x, i::Integer=1)
unsafe_store!(p::Ptr{T}, x, order::Symbol)
unsafe_store!(p::Ptr{T}, x, i::Integer, order::Symbol)
```

Store a value of type T to the address of the `i`th element (1-indexed) starting at `p`. This is equivalent to the C expression `p[i-1] = x`. Optionally, an atomic memory ordering can be provided.

The `unsafe` prefix on this function indicates that no validation is performed on the pointer `p` to ensure that it is valid. Like C, the programmer is responsible for ensuring that referenced memory is not freed or garbage collected while invoking this function. Incorrect usage may segfault your program. Unlike C, storing memory region allocated as different type may be valid provided that the types are compatible.

Julia 1.10

The `order` argument is available as of Julia 1.10.

See also: atomic

source

Base.unsafe_modify! – Function.

```
unsafe_modify!(p::Ptr{T}, op, x, [order::Symbol])::Pair
```

These atomically perform the operations to get and set a memory address after applying the function op. If supported by the hardware (for example, atomic increment), this may be optimized to the appropriate hardware instruction, otherwise its execution will be similar to:

```
y = unsafe_load(p)
z = op(y, x)
unsafe_store!(p, z)
return y => z
```

The unsafe prefix on this function indicates that no validation is performed on the pointer p to ensure that it is valid. Like C, the programmer is responsible for ensuring that referenced memory is not freed or garbage collected while invoking this function. Incorrect usage may segfault your program.

Julia 1.10

This function requires at least Julia 1.10.

See also: modifyproperty!, atomic

source

Base.unsafe_replace! – Function.

```
unsafe_replace!(p::Ptr{T}, expected, desired,
                [success_order::Symbol[, fail_order::Symbol=success_order]]) -> (; old,
                ↪  success::Bool)
```

These atomically perform the operations to get and conditionally set a memory address to a given value. If supported by the hardware, this may be optimized to the appropriate hardware instruction, otherwise its execution will be similar to:

```
y = unsafe_load(p, fail_order)
ok = y === expected
if ok
    unsafe_store!(p, desired, success_order)
end
return (; old = y, success = ok)
```

The unsafe prefix on this function indicates that no validation is performed on the pointer p to ensure that it is valid. Like C, the programmer is responsible for ensuring that referenced memory is not freed or garbage collected while invoking this function. Incorrect usage may segfault your program.

Julia 1.10

This function requires at least Julia 1.10.

See also: replaceproperty!, atomic

source

Base.unsafe_swap! – Function.

```
unsafe_swap!(p::Ptr{T}, x, [order::Symbol])
```

These atomically perform the operations to simultaneously get and set a memory address. If supported by the hardware, this may be optimized to the appropriate hardware instruction, otherwise its execution will be similar to:

```
y = unsafe_load(p)
unsafe_store!(p, x)
return y
```

The unsafe prefix on this function indicates that no validation is performed on the pointer p to ensure that it is valid. Like C, the programmer is responsible for ensuring that referenced memory is not freed or garbage collected while invoking this function. Incorrect usage may segfault your program.

Julia 1.10

This function requires at least Julia 1.10.

See also: swapproperty!, atomic

source

Base.unsafe_copyto! – Method.

```
unsafe_copyto!(dest::Ptr{T}, src::Ptr{T}, N)
```

Copy N elements from a source pointer to a destination, with no checking. The size of an element is determined by the type of the pointers.

The unsafe prefix on this function indicates that no validation is performed on the pointers dest and src to ensure that they are valid. Incorrect usage may corrupt or segfault your program, in the same manner as C.

source

Base.unsafe_copyto! – Method.

```
unsafe_copyto!(dest::Array, doffs, src::Array, soffs, n)
```

Copy n elements from a source array to a destination, starting at the linear index soffs in the source and doffs in the destination (1-indexed).

The unsafe prefix on this function indicates that no validation is performed to ensure that n is inbounds on either array. Incorrect usage may corrupt or segfault your program, in the same manner as C.

source

Base.copyto! – Function.

```
copyto!(dest, Rdest::CartesianIndices, src, Rsrc::CartesianIndices) -> dest
```

Copy the block of `src` in the range of `Rsrc` to the block of `dest` in the range of `Rdest`. The sizes of the two regions must match.

Examples

```
julia> A = zeros(5, 5);

julia> B = [1 2; 3 4];

julia> Ainds = CartesianIndices((2:3, 2:3));

julia> Binds = CartesianIndices(B);

julia> copyto!(A, Ainds, B, Binds)
5×5 Matrix{Float64}:
 0.0  0.0  0.0  0.0  0.0
 0.0  1.0  2.0  0.0  0.0
 0.0  3.0  4.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0
```

source

```
copyto!(dest::AbstractArray, src) -> dest
```

Copy all elements from collection `src` to array `dest`, whose length must be greater than or equal to the length `n` of `src`. The first `n` elements of `dest` are overwritten, the other elements are left untouched.

See also `copy!`, `copy`.

Warning

Behavior can be unexpected when any mutated argument shares memory with any other argument.

Examples

```
julia> x = [1., 0., 3., 0., 5.];

julia> y = zeros(7);

julia> copyto!(y, x);

julia> y
7-element Vector{Float64}:
 1.0
 0.0
 3.0
 0.0
 5.0
 0.0
 0.0
```

source

```
copyto!(dest, doffs, src, soffs, n)
```

Copy `n` elements from collection `src` starting at the linear index `soffs`, to array `dest` starting at the index `doffs`. Return `dest`.

source

```
copyto!(B::AbstractMatrix, ir_dest::AbstractUnitRange, jr_dest::AbstractUnitRange,
        tM::AbstractChar,
        M::AbstractVecOrMat, ir_src::AbstractUnitRange, jr_src::AbstractUnitRange) -> B
```

Efficiently copy elements of matrix `M` to `B` conditioned on the character parameter `tM` as follows:

tM	Destination	Source
'N'	B[ir_dest, jr_dest]	M[ir_src, jr_src]
'T'	B[ir_dest, jr_dest]	transpose(M)[ir_src, jr_src]
'C'	B[ir_dest, jr_dest]	adjoint(M)[ir_src, jr_src]

The elements `B[ir_dest, jr_dest]` are overwritten. Furthermore, the index range parameters must satisfy `length(ir_dest) == length(ir_src)` and `length(jr_dest) == length(jr_src)`.

See also `copy_transpose!` and `copy_adjoint!`.

```
copyto!(dest::AbstractMatrix, src::UniformScaling)
```

Copies a `UniformScaling` onto a matrix.

Julia 1.1

In Julia 1.0 this method only supported a square destination matrix. Julia 1.1. added support for a rectangular matrix.

`Base.pointer` – Function.

```
pointer(array [, index])
```

Get the native address of an array or string, optionally at a given location `index`.

This function is "unsafe". Be careful to ensure that a Julia reference to `array` exists as long as this pointer will be used. The `GC.@preserve` macro should be used to protect the `array` argument from garbage collection within a given block of code.

Calling `Ref(array[, index])` is generally preferable to this function as it guarantees validity.

source

`Base.unsafe_wrap` – Method.

```
unsafe_wrap(Array, pointer::Ptr{T}, dims; own = false)
```

Wrap a Julia `Array` object around the data at the address given by `pointer`, without making a copy. The pointer element type `T` determines the array element type. `dims` is either an integer (for a 1d array)

or a tuple of the array dimensions. `own` optionally specifies whether Julia should take ownership of the memory, calling `free` on the pointer when the array is no longer referenced.

This function is labeled "unsafe" because it will crash if `pointer` is not a valid memory address to data of the requested length. Unlike `unsafe_load` and `unsafe_store!`, the programmer is responsible also for ensuring that the underlying data is not accessed through two arrays of different element type, similar to the strict aliasing rule in C.

source

`Base.pointer_from_objref` – Function.

```
pointer_from_objref(x)
```

Get the memory address of a Julia object as a `Ptr`. The existence of the resulting `Ptr` will not protect the object from garbage collection, so you must ensure that the object remains referenced for the whole time that the `Ptr` will be used.

This function may not be called on immutable objects, since they do not have stable memory addresses.

See also `unsafe_pointer_to_objref`.

source

`Base.unsafe_pointer_to_objref` – Function.

```
unsafe_pointer_to_objref(p::Ptr)
```

Convert a `Ptr` to an object reference. Assumes the pointer refers to a valid heap-allocated Julia object. If this is not the case, undefined behavior results, hence this function is considered "unsafe" and should be used with care.

See also `pointer_from_objref`.

source

`Base.disable_sigint` – Function.

```
disable_sigint(f::Function)
```

Disable Ctrl-C handler during execution of a function on the current task, for calling external code that may call julia code that is not interrupt safe. Intended to be called using `do` block syntax as follows:

```
disable_sigint() do
    # interrupt-unsafe code
    ...
end
```

This is not needed on worker threads (`Threads.threadid() != 1`) since the `InterruptException` will only be delivered to the master thread. External functions that do not call julia code or julia runtime automatically disable sigint during their execution.

source

`Base.reenable_sigint` – Function.

```
reenable_sigint(f::Function)
```

Re-enable Ctrl-C handler during execution of a function. Temporarily reverses the effect of `disable_sigint`.

source

`Base.exit_on_sigint` – Function.

```
exit_on_sigint(on::Bool)
```

Set `exit_on_sigint` flag of the julia runtime. If `false`, Ctrl-C (SIGINT) is capturable as `InterruptException` in `try` block. This is the default behavior in REPL, any code run via `-e` and `-E` and in Julia script run with `-i` option.

If `true`, `InterruptException` is not thrown by Ctrl-C. Running code upon such event requires `atexit`. This is the default behavior in Julia script run without `-i` option.

Julia 1.5

Function `exit_on_sigint` requires at least Julia 1.5.

source

`Base.systemerror` – Function.

```
systemerror(sysfunc[, errno::Cint=Libc.errno()])
systemerror(sysfunc, iftrue::Bool)
```

Raises a `SystemError` for `errno` with the descriptive string `sysfunc` if `iftrue` is `true`

source

`Base.windowserror` – Function.

```
windowserror(sysfunc[, code::UInt32=Libc.GetLastError()])
windowserror(sysfunc, iftrue::Bool)
```

Like `systemerror`, but for Windows API functions that use `GetLastError` to return an error code instead of setting `errno`.

source

`Core.Ptr` – Type.

```
Ptr{T}
```

A memory address referring to data of type `T`. However, there is no guarantee that the memory is actually valid, or that it actually represents data of the specified type.

source

`Core.Ref` – Type.

```
Ref{T}
```

An object that safely references data of type `T`. This type is guaranteed to point to valid, Julia-allocated memory of the correct type. The underlying data is protected from freeing by the garbage collector as long as the `Ref` itself is referenced.

In Julia, `Ref` objects are dereferenced (loaded or stored) with `[]`.

Creation of a `Ref` to a value `x` of type `T` is usually written `Ref(x)`. Additionally, for creating interior pointers to containers (such as Array or Ptr), it can be written `Ref(a, i)` for creating a reference to the `i`-th element of `a`.

`Ref{T}()` creates a reference to a value of type `T` without initialization. For a bitstype `T`, the value will be whatever currently resides in the memory allocated. For a non-bitstype `T`, the reference will be undefined and attempting to dereference it will result in an error, "UndefRefError: access to undefined reference".

To check if a `Ref` is an undefined reference, use `isassigned(ref::RefValue)`. For example, `isassigned(Ref{T}())` is `false` if `T` is not a bitstype. If `T` is a bitstype, `isassigned(Ref{T}())` will always be true.

When passed as a `ccall` argument (either as a `Ptr` or `Ref` type), a `Ref` object will be converted to a native pointer to the data it references. For most `T`, or when converted to a `Ptr{Cvoid}`, this is a pointer to the object data. When `T` is an `isbits` type, this value may be safely mutated, otherwise mutation is strictly undefined behavior.

As a special case, setting `T = Any` will instead cause the creation of a pointer to the reference itself when converted to a `Ptr{Any}` (a `jl_value_t const* const*` if `T` is immutable, else a `jl_value_t *const *`). When converted to a `Ptr{Cvoid}`, it will still return a pointer to the data region as for any other `T`.

A `C_NULL` instance of `Ptr` can be passed to a `ccall` `Ref` argument to initialize it.

Use in broadcasting

`Ref` is sometimes used in broadcasting in order to treat the referenced values as a scalar.

Examples

```
julia> r = Ref(5) # Create a Ref with an initial value
Base.RefValue{Int64}(5)

julia> r[] # Getting a value from a Ref
5

julia> r[] = 7 # Storing a new value in a Ref
7

julia> r # The Ref now contains 7
Base.RefValue{Int64}(7)

julia> isa.(Ref([1,2,3]), [Array, Dict, Int]) # Treat reference values as scalar during
↪  broadcasting
3-element BitVector:
 1
 0
 0

julia> Ref{Function}()  # Undefined reference to a non-bitstype, Function
Base.RefValue{Function}(#undef)

julia> try
           Ref{Function}()[] # Dereferencing an undefined reference will result in an error
```

```
        catch e
            println(e)
        end
UndefRefError()

julia> Ref{Int64}()[]; # A reference to a bitstype refers to an undetermined value if not
↪  given

julia> isassigned(Ref{Int64}()) # A reference to a bitstype is always assigned
true
```

source

Base.isassigned – Method.

```
isassigned(ref::RefValue)::Bool
```

Test whether the given Ref is associated with a value. This is always true for a Ref of a bitstype object. Return false if the reference is undefined.

Examples

```
julia> ref = Ref{Function}()
Base.RefValue{Function}(#undef)

julia> isassigned(ref)
false

julia> ref[] = (foobar(x) = x)
foobar (generic function with 1 method)

julia> isassigned(ref)
true

julia> isassigned(Ref{Int}())
true
```

source

Base.Cchar – Type.

```
Cchar
```

Equivalent to the native char c-type.

source

Base.Cuchar – Type.

```
Cuchar
```

Equivalent to the native unsigned char c-type (UInt8).

source

`Base.Cshort` – Type.

```
Cshort
```

Equivalent to the native `signed short` c-type (`Int16`).

source

`Base.Cstring` – Type.

```
Cstring
```

A C-style string composed of the native character type `Cchar`s. `Cstring`s are NUL-terminated. For C-style strings composed of the native wide character type, see `Cwstring`. For more information about string interoperability with C, see the manual.

source

`Base.Cushort` – Type.

```
Cushort
```

Equivalent to the native `unsigned short` c-type (`UInt16`).

source

`Base.Cint` – Type.

```
Cint
```

Equivalent to the native `signed int` c-type (`Int32`).

source

`Base.Cuint` – Type.

```
Cuint
```

Equivalent to the native `unsigned int` c-type (`UInt32`).

source

`Base.Clong` – Type.

```
Clong
```

Equivalent to the native `signed long` c-type.

source

`Base.Culong` – Type.

```
Culong
```

Equivalent to the native `unsigned long` c-type.

source

`Base.Clonglong` – Type.

```
Clonglong
```

Equivalent to the native `signed long long` c-type (`Int64`).

source

`Base.Culonglong` – Type.

```
Culonglong
```

Equivalent to the native `unsigned long long` c-type (`UInt64`).

source

`Base.Cintmax_t` – Type.

```
Cintmax_t
```

Equivalent to the native `intmax_t` c-type (`Int64`).

source

`Base.Cuintmax_t` – Type.

```
Cuintmax_t
```

Equivalent to the native `uintmax_t` c-type (`UInt64`).

source

`Base.Csize_t` – Type.

```
Csize_t
```

Equivalent to the native `size_t` c-type (`UInt`).

source

`Base.Cssize_t` – Type.

```
Cssize_t
```

Equivalent to the native `ssize_t` c-type.

source

`Base.Cptrdiff_t` – Type.

```
Cptrdiff_t
```

Equivalent to the native `ptrdiff_t` c-type (`Int`).

source

`Base.Cwchar_t` – Type.

```
Cwchar_t
```

Equivalent to the native `wchar_t` c-type (`Int32`).

source

`Base.Cwstring` – Type.

```
Cwstring
```

A C-style string composed of the native wide character type `Cwchar_t`s. `Cwstring`s are NUL-terminated. For C-style strings composed of the native character type, see `Cstring`. For more information about string interoperability with C, see the manual.

source

`Base.Cfloat` – Type.

```
Cfloat
```

Equivalent to the native `float` c-type (`Float32`).

source

`Base.Cdouble` – Type.

```
Cdouble
```

Equivalent to the native `double` c-type (`Float64`).

source

Chapter 61

LLVM Interface

`Core.Intrinsics.llvmcall` – Function.

```
llvmcall(fun_ir::String, returntype, Tuple{argtype1, ...}, argvalue1, ...)
llvmcall((mod_ir::String, entry_fn::String), returntype, Tuple{argtype1, ...}, argvalue1, ...)
llvmcall((mod_bc::Vector{UInt8}, entry_fn::String), returntype, Tuple{argtype1, ...},
↪  argvalue1, ...)
```

Call the LLVM code provided in the first argument. There are several ways to specify this first argument:

- as a literal string, representing function-level IR (similar to an LLVM `define` block), with arguments are available as consecutive unnamed SSA variables (%0, %1, etc.);
- as a 2-element tuple, containing a string of module IR and a string representing the name of the entry-point function to call;
- as a 2-element tuple, but with the module provided as an `Vector{UInt8}` with bitcode.

Note that contrary to `ccall`, the argument types must be specified as a tuple type, and not a tuple of types. All types, as well as the LLVM code, should be specified as literals, and not as variables or expressions (it may be necessary to use `@eval` to generate these literals).

See `test/llvmcall.jl` for usage examples.

`source`

Chapter 62

C Standard Library

`Base.Libc.malloc` – Function.

```
malloc(size::Integer)::Ptr{Cvoid}
```

Call `malloc` from the C standard library.

source

`Base.Libc.calloc` – Function.

```
calloc(num::Integer, size::Integer)::Ptr{Cvoid}
```

Call `calloc` from the C standard library.

source

`Base.Libc.realloc` – Function.

```
realloc(addr::Ptr, size::Integer)::Ptr{Cvoid}
```

Call `realloc` from the C standard library.

See warning in the documentation for `free` regarding only using this on memory originally obtained from `malloc`.

source

`Base.memcpy` – Function.

```
memcpy(dst::Ptr, src::Ptr, n::Integer)::Ptr{Cvoid}
```

Call `memcpy` from the C standard library.

Julia 1.10

Support for `memcpy` requires at least Julia 1.10.

source

`Base.memmove` – Function.

```
memmove(dst::Ptr, src::Ptr, n::Integer)::Ptr{Cvoid}
```

Call memmove from the C standard library.

Julia 1.10

Support for memmove requires at least Julia 1.10.

source

Base.memset – Function.

```
memset(dst::Ptr, val, n::Integer)::Ptr{Cvoid}
```

Call memset from the C standard library.

Julia 1.10

Support for memset requires at least Julia 1.10.

source

Base.memcmp – Function.

```
memcmp(a::Ptr, b::Ptr, n::Integer)::Int
```

Call memcmp from the C standard library.

Julia 1.10

Support for memcmp requires at least Julia 1.9.

source

Base.Libc.free – Function.

```
free(addr::Ptr)
```

Call free from the C standard library. Only use this on memory obtained from malloc, not on pointers retrieved from other C libraries. Ptr objects obtained from C libraries should be freed by the free functions defined in that library, to avoid assertion failures if multiple libc libraries exist on the system.

source

Base.Libc.errno – Function.

```
errno([code])
```

Get the value of the C library's errno. If an argument is specified, it is used to set the value of errno.

The value of errno is only valid immediately after a ccall to a C library routine that sets it. Specifically, you cannot call errno at the next prompt in a REPL, because lots of code is executed between prompts.

source

Base.Libc.strerror – Function.

```
strerror(n=errno())
```

Convert a system call error code to a descriptive string

source

Base.Libc.GetLastError – Function.

```
GetLastError()
```

Call the Win32 GetLastError function [only available on Windows].

source

Base.Libc.FormatMessage – Function.

```
FormatMessage(n=GetLastError())
```

Convert a Win32 system call error code to a descriptive string [only available on Windows].

source

Base.Libc.time – Method.

```
time(t::TmStruct)::Float64
```

Converts a TmStruct struct to a number of seconds since the epoch.

source

Base.Libc.strftime – Function.

```
strftime([format], time)
```

Convert time, given as a number of seconds since the epoch or a TmStruct, to a formatted string using the given format. Supported formats are the same as those in the standard C library.

source

Base.Libc.strptime – Function.

```
strptime([format], timestr)
```

Parse a formatted time string into a TmStruct giving the seconds, minute, hour, date, etc. Supported formats are the same as those in the standard C library. On some platforms, timezones will not be parsed correctly. If the result of this function will be passed to time to convert it to seconds since the epoch, the isdst field should be filled in manually. Setting it to -1 will tell the C library to use the current system settings to determine the timezone.

source

Base.Libc.TmStruct – Type.

```
TmStruct([seconds])
```

Convert a number of seconds since the epoch to broken-down format, with fields sec, min, hour, mday, month, year, wday, yday, and isdst.

source

Base.Libc.FILE – Type.

```
FILE(::Ptr)
FILE(::IO)
```

A libc FILE*, representing an opened file.

It can be passed as a Ptr{FILE} argument to ccall and also supports seek, position and close.

A FILE can be constructed from an ordinary IO object, provided it is an open file. It must be closed afterward.

Examples

```
julia> using Base.Libc

julia> mktemp() do _, io
           # write to the temporary file using `puts(char*, FILE*)` from libc
           file = FILE(io)
           ccall(:fputs, Cint, (Cstring, Ptr{FILE}), "hello world", file)
           close(file)
           # read the file again
           seek(io, 0)
           read(io, String)
       end
"hello world"
```

source

Base.Libc.dup – Function.

```
dup(src::RawFD[, target::RawFD])::RawFD
```

Duplicate the file descriptor src so that the duplicate refers to the same OS resource (e.g. a file or socket). A target file descriptor may optionally be passed to use for the new duplicate.

source

Base.Libc.flush_cstdio – Function.

```
flush_cstdio()
```

Flushes the C stdout and stderr streams (which may have been written to by external C code).

source

Base.Libc.systemsleep – Function.

```
systemsleep(s::Real)
```

Suspends execution for s seconds. This function does not yield to Julia's scheduler and therefore blocks the Julia thread that it is running on for the duration of the sleep time.

See also sleep.

source

`Base.Libc.mkfifo` – Function.

```
mkfifo(path::AbstractString, [mode::Integer]) -> path
```

Make a FIFO special file (a named pipe) at `path`. Return `path` as-is on success.

`mkfifo` is supported only in Unix platforms.

Julia 1.11

`mkfifo` requires at least Julia 1.11.

source

Chapter 63

StackTraces

`Base.StackTraces.StackFrame` – Type.

```
StackFrame
```

Stack information representing execution context, with the following fields:

- `func::Symbol`
  The name of the function containing the execution context.
- `linfo::Union{Method, Core.MethodInstance, Core.CodeInstance, Core.CodeInfo, Nothing}`
  The Method, MethodInstance, CodeInstance, or CodeInfo containing the execution context (if it could be found), or nothing (for example, if the inlining was a result of macro expansion).
- `file::Symbol`
  The path to the file containing the execution context.
- `line::Int`
  The line number in the file containing the execution context.
- `from_c::Bool`
  True if the code is from C.
- `inlined::Bool`
  True if the code is from an inlined frame.
- `pointer::UInt64`
  Representation of the pointer to the execution context as returned by `backtrace`.

source

`Base.StackTraces.StackTrace` – Type.

```
StackTrace
```

An alias for `Vector{StackFrame}` provided for convenience; returned by calls to `stacktrace`.

source

`Base.StackTraces.stacktrace` – Function.

```
stacktrace([trace::Vector{Ptr{Cvoid}},] [c_funcs::Bool=false])::StackTrace
```

Return a stack trace in the form of a vector of StackFrames. (By default stacktrace doesn't return C functions, but this can be enabled.) When called without specifying a trace, stacktrace first calls backtrace.

source

The following methods and types in Base.StackTraces are not exported and need to be called e.g. as StackTraces.lookup(ptr).

Base.StackTraces.lookup – Function.

```
lookup(pointer::Ptr{Cvoid})::Vector{StackFrame}
```

Given a pointer to an execution context (usually generated by a call to backtrace), looks up stack frame context information. Returns an array of frame information for all functions inlined at that point, innermost function first.

source

Base.StackTraces.remove_frames! – Function.

```
remove_frames!(stack::StackTrace, m::Module)
```

Return the StackTrace with all StackFrames from the provided Module removed.

source

```
remove_frames!(stack::StackTrace, name::Symbol)
```

Takes a StackTrace (a vector of StackFrames) and a function name (a Symbol) and removes the StackFrame specified by the function name from the StackTrace (also removing all frames above the specified function). Primarily used to remove StackTraces functions from the StackTrace prior to returning it.

source

Chapter 64

SIMD Support

Type `VecElement{T}` is intended for building libraries of SIMD operations. Practical use of it requires using `llvmcall`. The type is defined as:

```
struct VecElement{T}
    value::T
end
```

It has a special compilation rule: a homogeneous tuple of `VecElement{T}` maps to an LLVM `vector` type when T is a primitive bits type.

At `-O3`, the compiler *might* automatically vectorize operations on such tuples. For example, the following program, when compiled with `julia -O3` generates two SIMD addition instructions (`addps`) on x86 systems:

```
const m128 = NTuple{4,VecElement{Float32}}

function add(a::m128, b::m128)
    (VecElement(a[1].value+b[1].value),
     VecElement(a[2].value+b[2].value),
     VecElement(a[3].value+b[3].value),
     VecElement(a[4].value+b[4].value))
end

triple(c::m128) = add(add(c,c),c)

code_native(triple,(m128,))
```

However, since the automatic vectorization cannot be relied upon, future use will mostly be via libraries that use `llvmcall`.

Part III

Standard Library

Chapter 65

ArgTools

65.1 Argument Handling

`ArgTools.ArgRead` – Type.

```
ArgRead = Union{AbstractString, AbstractCmd, IO}
```

The `ArgRead` types is a union of the types that the `arg_read` function knows how to convert into readable IO handles. See `arg_read` for details.

`ArgTools.ArgWrite` – Type.

```
ArgWrite = Union{AbstractString, AbstractCmd, IO}
```

The `ArgWrite` types is a union of the types that the `arg_write` function knows how to convert into writeable IO handles, except for `Nothing` which `arg_write` handles by generating a temporary file. See `arg_write` for details.

`ArgTools.arg_read` – Function.

```
arg_read(f::Function, arg::ArgRead) -> f(arg_io)
```

The `arg_read` function accepts an argument `arg` that can be any of these:

- `AbstractString`: a file path to be opened for reading
- `AbstractCmd`: a command to be run, reading from its standard output
- `IO`: an open IO handle to be read from

Whether the body returns normally or throws an error, a path which is opened will be closed before returning from `arg_read` and an `IO` handle will be flushed but not closed before returning from `arg_read`.

Note: when opening a file, ArgTools will pass `lock = false` to the file `open(...)` call. Therefore, the object returned by this function should not be used from multiple threads. This restriction may be relaxed in the future, which would not break any working code.

`ArgTools.arg_write` – Function.

```
arg_write(f::Function, arg::ArgWrite) -> arg
arg_write(f::Function, arg::Nothing) -> tempname()
```

The `arg_write` function accepts an argument `arg` that can be any of these:

- `AbstractString`: a file path to be opened for writing
- `AbstractCmd`: a command to be run, writing to its standard input
- `IO`: an open IO handle to be written to
- `Nothing`: a temporary path should be written to

If the body returns normally, a path that is opened will be closed upon completion; an IO handle argument is left open but flushed before return. If the argument is `nothing` then a temporary path is opened for writing and closed open completion and the path is returned from `arg_write`. In all other cases, `arg` itself is returned. This is a useful pattern since you can consistently return whatever was written, whether an argument was passed or not.

If there is an error during the evaluation of the body, a path that is opened by `arg_write` for writing will be deleted, whether it's passed in as a string or a temporary path generated when `arg` is `nothing`.

Note: when opening a file, ArgTools will pass `lock = false` to the file `open(...)` call. Therefore, the object returned by this function should not be used from multiple threads. This restriction may be relaxed in the future, which would not break any working code.

`ArgTools.arg_isdir` – Function.

```
arg_isdir(f::Function, arg::AbstractString) -> f(arg)
```

The `arg_isdir` function takes `arg` which must be the path to an existing directory (an error is raised otherwise) and passes that path to `f` finally returning the result of `f(arg)`. This is definitely the least useful tool offered by `ArgTools` and mostly exists for symmetry with `arg_mkdir` and to give consistent error messages.

`ArgTools.arg_mkdir` – Function.

```
arg_mkdir(f::Function, arg::AbstractString) -> arg
arg_mkdir(f::Function, arg::Nothing) -> mktempdir()
```

The `arg_mkdir` function takes `arg` which must either be one of:

- a path to an already existing empty directory,
- a non-existent path which can be created as a directory, or
- `nothing` in which case a temporary directory is created.

In all cases the path to the directory is returned. If an error occurs during `f(arg)`, the directory is returned to its original state: if it already existed but was empty, it will be emptied; if it did not exist it will be deleted.

65.2 Function Testing

`ArgTools.arg_readers` – Function.

```
arg_readers(arg :: AbstractString, [ type = ArgRead ]) do arg::Function
    ## pre-test setup ##
    @arg_test arg begin
        arg :: ArgRead
        ## test using `arg` ##
    end
    ## post-test cleanup ##
end
```

The `arg_readers` function takes a path to be read and a single-argument do block, which is invoked once for each test reader type that `arg_read` can handle. If the optional `type` argument is given then the do block is only invoked for readers that produce arguments of that type.

The `arg` passed to the do block is not the argument value itself, because some of test argument types need to be initialized and finalized for each test case. Consider an open file handle argument: once you've used it for one test, you can't use it again; you need to close it and open the file again for the next test. This function `arg` can be converted into an `ArgRead` instance using `@arg_test arg begin ... end`.

`ArgTools.arg_writers` – Function.

```
arg_writers([ type = ArgWrite ]) do path::String, arg::Function
    ## pre-test setup ##
    @arg_test arg begin
        arg :: ArgWrite
        ## test using `arg` ##
    end
    ## post-test cleanup ##
end
```

The `arg_writers` function takes a do block, which is invoked once for each test writer type that `arg_write` can handle with a temporary (non-existent) `path` and `arg` which can be converted into various writable argument types which write to `path`. If the optional `type` argument is given then the do block is only invoked for writers that produce arguments of that type.

The `arg` passed to the do block is not the argument value itself, because some of test argument types need to be initialized and finalized for each test case. Consider an open file handle argument: once you've used it for one test, you can't use it again; you need to close it and open the file again for the next test. This function `arg` can be converted into an `ArgWrite` instance using `@arg_test arg begin ... end`.

There is also an `arg_writers` method that takes a path name like `arg_readers`:

```
arg_writers(path::AbstractString, [ type = ArgWrite ]) do arg::Function
    ## pre-test setup ##
    @arg_test arg begin
        # here `arg :: ArgWrite`
        ## test using `arg` ##
    end
    ## post-test cleanup ##
end
```

This method is useful if you need to specify `path` instead of using path name generated by `tempname()`. Since `path` is passed from outside of `arg_writers`, the path is not an argument to the do block in this form.

`ArgTools.@arg_test` – Macro.

```
@arg_test arg1 arg2 ... body
```

The `@arg_test` macro is used to convert `arg` functions provided by `arg_readers` and `arg_writers` into actual argument values. When you write `@arg_test arg body` it is equivalent to `arg(arg -> body)`.

Chapter 66

Artifacts

Starting with Julia 1.6, the artifacts support has moved from `Pkg.jl` to Julia itself. Until proper documentation can be added here, you can learn more about artifacts in the `Pkg.jl` manual at https://julialang.github.io/Pkg.jl/v1/artifacts/.

Julia 1.6

Julia's artifacts API requires at least Julia 1.6. In Julia versions 1.3 to 1.5, you can use `Pkg.Artifacts` instead.

`Artifacts.artifact_meta` – Function.

```
artifact_meta(name::String, artifacts_toml::String;
              platform::AbstractPlatform = HostPlatform(),
              pkg_uuid::Union{Base.UUID,Nothing}=nothing)
```

Get metadata about a given artifact (identified by name) stored within the given `(Julia)Artifacts.toml` file. If the artifact is platform-specific, use `platform` to choose the most appropriate mapping. If none is found, return `nothing`.

Julia 1.3

This function requires at least Julia 1.3.

`Artifacts.artifact_hash` – Function.

```
artifact_hash(name::String, artifacts_toml::String;
              platform::AbstractPlatform = HostPlatform())
```

Thin wrapper around `artifact_meta()` to return the hash of the specified, platform- collapsed artifact. Returns `nothing` if no mapping can be found.

Julia 1.3

This function requires at least Julia 1.3.

`Artifacts.find_artifacts_toml` – Function.

```
find_artifacts_toml(path::String)
```

Given the path to a `.jl` file, (such as the one returned by `__source__.file` in a macro context), find the `(Julia)Artifacts.toml` that is contained within the containing project (if it exists), otherwise return `nothing`.

Julia 1.3

This function requires at least Julia 1.3.

`Artifacts.@artifact_str` – Macro.

```
macro artifact_str(name)
```

Return the on-disk path to an artifact. Automatically looks the artifact up by name in the project's `(Julia)Artifacts.toml` file. Throws an error on if the requested artifact is not present. If run in the REPL, searches for the toml file starting in the current directory, see `find_artifacts_toml()` for more.

If the artifact is marked "lazy" and the package has `using LazyArtifacts` defined, the artifact will be downloaded on-demand with Pkg the first time this macro tries to compute the path. The files will then be left installed locally for later.

If `name` contains a forward or backward slash, all elements after the first slash will be taken to be path names indexing into the artifact, allowing for an easy one-liner to access a single file/directory within an artifact. Example:

```
ffmpeg_path = @artifact"FFMPEG/bin/ffmpeg"
```

Julia 1.3

This macro requires at least Julia 1.3.

Julia 1.6

Slash-indexing requires at least Julia 1.6.

`Artifacts.artifact_exists` – Function.

```
artifact_exists(hash::SHA1; honor_overrides::Bool=true)
```

Return whether or not the given artifact (identified by its sha1 git tree hash) exists on-disk. Note that it is possible that the given artifact exists in multiple locations (e.g. within multiple depots).

Julia 1.3

This function requires at least Julia 1.3.

`Artifacts.artifact_path` – Function.

```
artifact_path(hash::SHA1; honor_overrides::Bool=true)
```

Given an artifact (identified by SHA1 git tree hash), return its installation path. If the artifact does not exist, returns the location it would be installed to.

Julia 1.3

This function requires at least Julia 1.3.

`Artifacts.select_downloadable_artifacts` – Function.

```
select_downloadable_artifacts(artifacts_toml::String;
                              platform = HostPlatform,
                              include_lazy = false,
                              pkg_uuid = nothing)
```

Return a dictionary where every entry is an artifact from the given `Artifacts.toml` that should be downloaded for the requested platform. Lazy artifacts are included if `include_lazy` is set.

Chapter 67

Base64

Base64.Base64 – Module.

```
Base64
```

Functionality for base64 encoding and decoding, a method to represent binary data using text, common on the web.

Base64.Base64EncodePipe – Type.

```
Base64EncodePipe(ostream)
```

Return a new write-only I/O stream, which converts any bytes written to it into base64-encoded ASCII bytes written to `ostream`. Calling `close` on the `Base64EncodePipe` stream is necessary to complete the encoding (but does not close `ostream`).

Examples

```
julia> io = IOBuffer();

julia> iob64_encode = Base64EncodePipe(io);

julia> write(iob64_encode, "Hello!")
6

julia> close(iob64_encode);

julia> str = takestring!(io)
"SGVsbG8h"

julia> String(base64decode(str))
"Hello!"
```

Base64.base64encode – Function.

```
base64encode(writefunc, args...; context=nothing)
base64encode(args...; context=nothing)
```

Given a write-like function writefunc, which takes an I/O stream as its first argument, base64encode(writefunc, args...) calls writefunc to write args... to a base64-encoded string, and returns the string. base64encode(args...) is equivalent to base64encode(write, args...): it converts its arguments into bytes using the standard write functions and returns the base64-encoded string.

The optional keyword argument context can be set to :key=>value pair or an IO or IOContext object whose attributes are used for the I/O stream passed to writefunc or write.

See also base64decode.

Base64.Base64DecodePipe – Type.

```
Base64DecodePipe(istream)
```

Return a new read-only I/O stream, which decodes base64-encoded data read from istream.

Examples

```
julia> io = IOBuffer();

julia> iob64_decode = Base64DecodePipe(io);

julia> write(io, "SGVsbG8h")
8

julia> seekstart(io);

julia> String(read(iob64_decode))
"Hello!"
```

Base64.base64decode – Function.

```
base64decode(string)
```

Decode the base64-encoded string and returns a Vector{UInt8} of the decoded bytes.

See also base64encode.

Examples

```
julia> b = base64decode("SGVsbG8h")
6-element Vector{UInt8}:
 0x48
 0x65
 0x6c
 0x6c
 0x6f
 0x21

julia> String(b)
"Hello!"
```

Base64.stringmime – Function.

```
stringmime(mime, x; context=nothing)
```

Return an `AbstractString` containing the representation of `x` in the requested `mime` type. This is similar to `repr(mime, x)` except that binary data is base64-encoded as an ASCII string.

The optional keyword argument `context` can be set to `:key=>value` pair or an `IO` or `IOContext` object whose attributes are used for the I/O stream passed to `show`.

Chapter 68

CRC32c

Standard library module for computing the CRC-32c checksum.

CRC32c.crc32c – Function.

```
crc32c(data, crc::UInt32=0x00000000)
```

Compute the CRC-32c checksum of the given `data`, which can be an `Array{UInt8}`, a contiguous subarray thereof, an `AbstractVector{UInt8}`, or a `String`. Optionally, you can pass a starting `crc` integer to be mixed in with the checksum. The `crc` parameter can be used to compute a checksum on data divided into chunks: performing `crc32c(data2, crc32c(data1))` is equivalent to the checksum of `[data1; data2]`. (Technically, a little-endian checksum is computed.)

There is also a method `crc32c(io, nb, crc)` to checksum `nb` bytes from a stream `io`, or `crc32c(io, crc)` to checksum all the remaining bytes. Hence you can do `open(crc32c, filename)` to checksum an entire file, or `crc32c(seekstart(buf))` to checksum an `IOBuffer` without calling `take!`.

For a `String`, note that the result is specific to the UTF-8 encoding (a different checksum would be obtained from a different Unicode encoding). To checksum an `a::AbstractArray` of some other bitstype without padding, you can do `crc32c(vec(reinterpret(UInt8,a)))`, but note that the result may be endian-dependent.

CRC32c.crc32c – Method.

```
crc32c(io::IO, [nb::Integer,] crc::UInt32=0x00000000)
```

Read up to `nb` bytes from `io` and return the CRC-32c checksum, optionally mixed with a starting `crc` integer. If `nb` is not supplied, then `io` will be read until the end of the stream.

Chapter 69

Dates

The `Dates` module provides two types for working with dates: `Date` and `DateTime`, representing day and millisecond precision, respectively; both are subtypes of the abstract `TimeType`. The motivation for distinct types is simple: some operations are much simpler, both in terms of code and mental reasoning, when the complexities of greater precision don't have to be dealt with. For example, since the `Date` type only resolves to the precision of a single date (i.e. no hours, minutes, or seconds), normal considerations for time zones, daylight savings/summer time, and leap seconds are unnecessary and avoided.

Both `Date` and `DateTime` are basically immutable `Int64` wrappers. The single `instant` field of either type is actually a `UTInstant{P}` type, which represents a continuously increasing machine timeline based on the UT second [1]. The `DateTime` type is not aware of time zones (*naive*, in Python parlance), analogous to a *LocalDateTime* in Java 8. Additional time zone functionality can be added through the TimeZones.jl package, which compiles the IANA time zone database. Both `Date` and `DateTime` are based on the ISO 8601 standard, which follows the proleptic Gregorian calendar. One note is that the ISO 8601 standard is particular about BC/BCE dates. In general, the last day of the BC/BCE era, 1-12-31 BC/BCE, was followed by 1-1-1 AD/CE, thus no year zero exists. The ISO standard, however, states that 1 BC/BCE is year zero, so `0000-12-31` is the day before `0001-01-01`, and year `-0001` (yes, negative one for the year) is 2 BC/BCE, year `-0002` is 3 BC/BCE, etc.

69.1 Constructors

`Date` and `DateTime` types can be constructed by integer or `Period` types, by parsing, or through adjusters (more on those later):

```
julia> DateTime(2013)
2013-01-01T00:00:00

julia> DateTime(2013,7)
2013-07-01T00:00:00
```

[1] The notion of the UT second is actually quite fundamental. There are basically two different notions of time generally accepted, one based on the physical rotation of the earth (one full rotation = 1 day), the other based on the SI second (a fixed, constant value). These are radically different! Think about it, a "UT second", as defined relative to the rotation of the earth, may have a different absolute length depending on the day! Anyway, the fact that `Date` and `DateTime` are based on UT seconds is a simplifying, yet honest assumption so that things like leap seconds and all their complexity can be avoided. This basis of time is formally called UT or UT1. Basing types on the UT second basically means that every minute has 60 seconds and every day has 24 hours and leads to more natural calculations when working with calendar dates.

```
julia> DateTime(2013,7,1)
2013-07-01T00:00:00

julia> DateTime(2013,7,1,12)
2013-07-01T12:00:00

julia> DateTime(2013,7,1,12,30)
2013-07-01T12:30:00

julia> DateTime(2013,7,1,12,30,59)
2013-07-01T12:30:59

julia> DateTime(2013,7,1,12,30,59,1)
2013-07-01T12:30:59.001

julia> Date(2013)
2013-01-01

julia> Date(2013,7)
2013-07-01

julia> Date(2013,7,1)
2013-07-01

julia> Date(Dates.Year(2013),Dates.Month(7),Dates.Day(1))
2013-07-01

julia> Date(Dates.Month(7),Dates.Year(2013))
2013-07-01
```

`Date` or `DateTime` parsing is accomplished by the use of format strings. Format strings work by the notion of defining *delimited* or *fixed-width* "slots" that contain a period to parse and passing the text to parse and format string to a `Date` or `DateTime` constructor, of the form `Date("2015-01-01",dateformat"y-m-d")` or `DateTime("20150101",dateformat"yyyymmdd")`.

Delimited slots are marked by specifying the delimiter the parser should expect between two subsequent periods; so `"y-m-d"` lets the parser know that between the first and second slots in a date string like `"2014-07-16"`, it should find the `-` character. The `y`, `m`, and `d` characters let the parser know which periods to parse in each slot.

As in the case of constructors above such as `Date(2013)`, delimited `DateFormats` allow for missing parts of dates and times so long as the preceding parts are given. The other parts are given the usual default values. For example, `Date("1981-03", dateformat"y-m-d")` returns `1981-03-01`, whilst `Date("31/12", dateformat"d/m/y")` gives `0001-12-31`. (Note that the default year is 1 AD/CE.) An empty string, however, always throws an `ArgumentError`.

Fixed-width slots are specified by repeating the period character the number of times corresponding to the width with no delimiter between characters. So `dateformat"yyyymmdd"` would correspond to a date string like `"20140716"`. The parser distinguishes a fixed-width slot by the absence of a delimiter, noting the transition `"yyyymm"` from one period character to the next.

Support for text-form month parsing is also supported through the `u` and `U` characters, for abbreviated and full-length month names, respectively. By default, only English month names are supported, so `u` corre-

sponds to "Jan", "Feb", "Mar", etc. And U corresponds to "January", "February", "March", etc. Similar to other name=>value mapping functions `dayname` and `monthname`, custom locales can be loaded by passing in the `locale=>Dict{String,Int}` mapping to the `MONTHTOVALUEABBR` and `MONTHTOVALUE` dicts for abbreviated and full-name month names, respectively.

The above examples used the `dateformat""` string macro. This macro creates a `DateFormat` object once when the macro is expanded and uses the same `DateFormat` object even if a code snippet is run multiple times.

```
julia> for i = 1:10^5
           Date("2015-01-01", dateformat"y-m-d")
       end
```

Or you can create the DateFormat object explicitly:

```
julia> df = DateFormat("y-m-d");

julia> dt = Date("2015-01-01",df)
2015-01-01

julia> dt2 = Date("2015-01-02",df)
2015-01-02
```

Alternatively, use broadcasting:

```
julia> years = ["2015", "2016"];

julia> Date.(years, DateFormat("yyyy"))
2-element Vector{Date}:
 2015-01-01
 2016-01-01
```

For convenience, you may pass the format string directly (e.g., `Date("2015-01-01","y-m-d")`), although this form incurs performance costs if you are parsing the same format repeatedly, as it internally creates a new `DateFormat` object each time.

As well as via the constructors, a `Date` or `DateTime` can be constructed from strings using the `parse` and `tryparse` functions, but with an optional third argument of type `DateFormat` specifying the format; for example, `parse(Date, "06.23.2013", dateformat"m.d.y")`, or `tryparse(DateTime, "1999-12-31T23:59:59")` which uses the default format. The notable difference between the functions is that with `tryparse`, an error is not thrown if the string is empty or in an invalid format; instead `nothing` is returned.

Julia 1.9

Before Julia 1.9, empty strings could be passed to constructors and `parse` without error, returning as appropriate `DateTime(1)`, `Date(1)` or `Time(0)`. Likewise, `tryparse` did not return `nothing`.

A full suite of parsing and formatting tests and examples is available in `stdlib/Dates/test/io.jl`.

69.2 Durations/Comparisons

Finding the length of time between two `Date` or `DateTime` is straightforward given their underlying representation as `UTInstant{Day}` and `UTInstant{Millisecond}`, respectively. The difference between `Date` is returned in the number of `Day`, and `DateTime` in the number of `Millisecond`. Similarly, comparing `TimeType` is a simple matter of comparing the underlying machine instants (which in turn compares the internal `Int64` values).

```
julia> dt = Date(2012,2,29)
2012-02-29

julia> dt2 = Date(2000,2,1)
2000-02-01

julia> dump(dt)
Date
  instant: Dates.UTInstant{Day}
    periods: Day
      value: Int64 734562

julia> dump(dt2)
Date
  instant: Dates.UTInstant{Day}
    periods: Day
      value: Int64 730151

julia> dt > dt2
true

julia> dt != dt2
true

julia> dt + dt2
ERROR: MethodError: no method matching +(::Date, ::Date)
[...]

julia> dt * dt2
ERROR: MethodError: no method matching *(::Date, ::Date)
[...]

julia> dt / dt2
ERROR: MethodError: no method matching /(::Date, ::Date)

julia> dt - dt2
4411 days

julia> dt2 - dt
-4411 days

julia> dt = DateTime(2012,2,29)
2012-02-29T00:00:00

julia> dt2 = DateTime(2000,2,1)
```

```
2000-02-01T00:00:00

julia> dt - dt2
381110400000 milliseconds
```

69.3 Accessor Functions

Because the `Date` and `DateTime` types are stored as single `Int64` values, date parts or fields can be retrieved through accessor functions. The lowercase accessors return the field as an integer:

```
julia> t = Date(2014, 1, 31)
2014-01-31

julia> Dates.year(t)
2014

julia> Dates.month(t)
1

julia> Dates.week(t)
5

julia> Dates.day(t)
31
```

While propercase return the same value in the corresponding `Period` type:

```
julia> Dates.Year(t)
2014 years

julia> Dates.Day(t)
31 days
```

Compound methods are provided because it is more efficient to access multiple fields at the same time than individually:

```
julia> Dates.yearmonth(t)
(2014, 1)

julia> Dates.monthday(t)
(1, 31)

julia> Dates.yearmonthday(t)
(2014, 1, 31)
```

One may also access the underlying `UTInstant` or integer value:

```
julia> dump(t)
Date
```

```
  instant: Dates.UTInstant{Day}
    periods: Day
      value: Int64 735264

julia> t.instant
Dates.UTInstant{Day}(Day(735264))

julia> Dates.value(t)
735264
```

69.4 Query Functions

Query functions provide calendrical information about a `TimeType`. They include information about the day of the week:

```
julia> t = Date(2014, 1, 31)
2014-01-31

julia> Dates.dayofweek(t)
5

julia> Dates.dayname(t)
"Friday"

julia> Dates.dayofweekofmonth(t) # 5th Friday of January
5
```

Month of the year:

```
julia> Dates.monthname(t)
"January"

julia> Dates.daysinmonth(t)
31
```

As well as information about the `TimeType`'s year and quarter:

```
julia> Dates.isleapyear(t)
false

julia> Dates.dayofyear(t)
31

julia> Dates.quarterofyear(t)
1

julia> Dates.dayofquarter(t)
31
```

The dayname and monthname methods can also take an optional locale keyword that can be used to return the name of the day or month of the year for other languages/locales. There are also versions of these functions returning the abbreviated names, namely dayabbr and monthabbr. First the mapping is loaded into the LOCALES variable:

```
julia> french_months = ["janvier", "février", "mars", "avril", "mai", "juin",
                        "juillet", "août", "septembre", "octobre", "novembre", "décembre"];

julia> french_months_abbrev = ["janv","févr","mars","avril","mai","juin",
                               "juil","août","sept","oct","nov","déc"];

julia> french_days = ["lundi","mardi","mercredi","jeudi","vendredi","samedi","dimanche"];

julia> Dates.LOCALES["french"] = Dates.DateLocale(french_months, french_months_abbrev,
↪  french_days, [""]);
```

The above mentioned functions can then be used to perform the queries:

```
julia> Dates.dayname(t;locale="french")
"vendredi"

julia> Dates.monthname(t;locale="french")
"janvier"

julia> Dates.monthabbr(t;locale="french")
"janv"
```

Since the abbreviated versions of the days are not loaded, trying to use the function dayabbr will throw an error.

```
julia> Dates.dayabbr(t;locale="french")
ERROR: BoundsError: attempt to access 1-element Vector{String} at index [5]
Stacktrace:
[...]
```

69.5 TimeType-Period Arithmetic

It's good practice when using any language/date framework to be familiar with how date-period arithmetic is handled as there are some tricky issues to deal with (though much less so for day-precision types).

The Dates module approach tries to follow the simple principle of trying to change as little as possible when doing Period arithmetic. This approach is also often known as *calendrical* arithmetic or what you would probably guess if someone were to ask you the same calculation in a conversation. Why all the fuss about this? Let's take a classic example: add 1 month to January 31st, 2014. What's the answer? Javascript will say March 3 (assumes 31 days). PHP says March 2 (assumes 30 days). The fact is, there is no right answer. In the Dates module, it gives the result of February 28th. How does it figure that out? Consider the classic 7-7-7 gambling game in casinos.

Now just imagine that instead of 7-7-7, the slots are Year-Month-Day, or in our example, 2014-01-31. When you ask to add 1 month to this date, the month slot is incremented, so now we have 2014-02-31. Then the

day number is checked if it is greater than the last valid day of the new month; if it is (as in the case above), the day number is adjusted down to the last valid day (28). What are the ramifications with this approach? Go ahead and add another month to our date, `2014-02-28 + Month(1) == 2014-03-28`. What? Were you expecting the last day of March? Nope, sorry, remember the 7-7-7 slots. As few slots as possible are going to change, so we first increment the month slot by 1, 2014-03-28, and boom, we're done because that's a valid date. On the other hand, if we were to add 2 months to our original date, 2014-01-31, then we end up with 2014-03-31, as expected. The other ramification of this approach is a loss in associativity when a specific ordering is forced (i.e. adding things in different orders results in different outcomes). For example:

```
julia> (Date(2014,1,29)+Dates.Day(1)) + Dates.Month(1)
2014-02-28

julia> (Date(2014,1,29)+Dates.Month(1)) + Dates.Day(1)
2014-03-01
```

What's going on there? In the first line, we're adding 1 day to January 29th, which results in 2014-01-30; then we add 1 month, so we get 2014-02-30, which then adjusts down to 2014-02-28. In the second example, we add 1 month *first*, where we get 2014-02-29, which adjusts down to 2014-02-28, and *then* add 1 day, which results in 2014-03-01. One design principle that helps in this case is that, in the presence of multiple Periods, the operations will be ordered by the Periods' *types*, not their value or positional order; this means `Year` will always be added first, then `Month`, then `Week`, etc. Hence the following *does* result in associativity and Just Works:

```
julia> Date(2014,1,29) + Dates.Day(1) + Dates.Month(1)
2014-03-01

julia> Date(2014,1,29) + Dates.Month(1) + Dates.Day(1)
2014-03-01
```

Tricky? Perhaps. What is an innocent `Dates` user to do? The bottom line is to be aware that explicitly forcing a certain associativity, when dealing with months, may lead to some unexpected results, but otherwise, everything should work as expected. Thankfully, that's pretty much the extent of the odd cases in date-period arithmetic when dealing with time in UT (avoiding the "joys" of dealing with daylight savings, leap seconds, etc.).

As a bonus, all period arithmetic objects work directly with ranges:

```
julia> dr = Date(2014,1,29):Day(1):Date(2014,2,3)
Date("2014-01-29"):Day(1):Date("2014-02-03")

julia> collect(dr)
6-element Vector{Date}:
 2014-01-29
 2014-01-30
 2014-01-31
 2014-02-01
 2014-02-02
 2014-02-03
```

```
julia> dr = Date(2014,1,29):Dates.Month(1):Date(2014,07,29)
Date("2014-01-29"):Month(1):Date("2014-07-29")

julia> collect(dr)
7-element Vector{Date}:
 2014-01-29
 2014-02-28
 2014-03-29
 2014-04-29
 2014-05-29
 2014-06-29
 2014-07-29
```

69.6 Adjuster Functions

As convenient as date-period arithmetic is, often the kinds of calculations needed on dates take on a *calendrical* or *temporal* nature rather than a fixed number of periods. Holidays are a perfect example; most follow rules such as "Memorial Day = Last Monday of May", or "Thanksgiving = 4th Thursday of November". These kinds of temporal expressions deal with rules relative to the calendar, like first or last of the month, next Tuesday, or the first and third Wednesdays, etc.

The `Dates` module provides the *adjuster* API through several convenient methods that aid in simply and succinctly expressing temporal rules. The first group of adjuster methods deal with the first and last of weeks, months, quarters, and years. They each take a single `TimeType` as input and return or *adjust to* the first or last of the desired period relative to the input.

```
julia> Dates.firstdayofweek(Date(2014,7,16)) # Adjusts the input to the Monday of the input's
↪  week
2014-07-14

julia> Dates.lastdayofmonth(Date(2014,7,16)) # Adjusts to the last day of the input's month
2014-07-31

julia> Dates.lastdayofquarter(Date(2014,7,16)) # Adjusts to the last day of the input's quarter
2014-09-30
```

The next two higher-order methods, `tonext`, and `toprev`, generalize working with temporal expressions by taking a `DateFunction` as first argument, along with a starting `TimeType`. A `DateFunction` is just a function, usually anonymous, that takes a single `TimeType` as input and returns a `Bool`, `true` indicating a satisfied adjustment criterion. For example:

```
julia> istuesday = x->Dates.dayofweek(x) == Dates.Tuesday; # Returns true if the day of the week
↪  of x is Tuesday

julia> Dates.tonext(istuesday, Date(2014,7,13)) # 2014-07-13 is a Sunday
2014-07-15

julia> Dates.tonext(Date(2014,7,13), Dates.Tuesday) # Convenience method provided for day of the
↪  week adjustments
2014-07-15
```

This is useful with the do-block syntax for more complex temporal expressions:

```
julia> Dates.tonext(Date(2014,7,13)) do x
           # Return true on the 4th Thursday of November (Thanksgiving)
           Dates.dayofweek(x) == Dates.Thursday &&
           Dates.dayofweekofmonth(x) == 4 &&
           Dates.month(x) == Dates.November
       end
2014-11-27
```

The `Base.filter` method can be used to obtain all valid dates/moments in a specified range:

```
# Pittsburgh street cleaning; Every 2nd Tuesday from April to November
# Date range from January 1st, 2014 to January 1st, 2015
julia> dr = Dates.Date(2014):Day(1):Dates.Date(2015);

julia> filter(dr) do x
           Dates.dayofweek(x) == Dates.Tue &&
           Dates.April <= Dates.month(x) <= Dates.Nov &&
           Dates.dayofweekofmonth(x) == 2
       end
8-element Vector{Date}:
 2014-04-08
 2014-05-13
 2014-06-10
 2014-07-08
 2014-08-12
 2014-09-09
 2014-10-14
 2014-11-11
```

Additional examples and tests are available in `stdlib/Dates/test/adjusters.jl`.

69.7 Period Types

Periods are a human view of discrete, sometimes irregular durations of time. Consider 1 month; it could represent, in days, a value of 28, 29, 30, or 31 depending on the year and month context. Or a year could represent 365 or 366 days in the case of a leap year. `Period` types are simple `Int64` wrappers and are constructed by wrapping any `Int64` convertible type, i.e. `Year(1)` or `Month(3.0)`. Arithmetic between `Period` of the same type behave like integers, and limited `Period-Real` arithmetic is available. You can extract the underlying integer with `Dates.value`.

```
julia> y1 = Dates.Year(1)
1 year

julia> y2 = Dates.Year(2)
2 years

julia> y3 = Dates.Year(10)
10 years
```

```
julia> y1 + y2
3 years

julia> div(y3,y2)
5

julia> y3 - y2
8 years

julia> y3 % y2
0 years

julia> div(y3,3) # mirrors integer division
3 years

julia> Dates.value(Dates.Millisecond(10))
10
```

Representing periods or durations that are not integer multiples of the basic types can be achieved with the `Dates.CompoundPeriod` type. Compound periods may be constructed manually from simple `Period` types. Additionally, the `canonicalize` function can be used to break down a period into a `Dates.CompoundPeriod`. This is particularly useful to convert a duration, e.g., a difference of two `DateTime`, into a more convenient representation.

```
julia> cp = Dates.CompoundPeriod(Day(1),Minute(1))
1 day, 1 minute

julia> t1 = DateTime(2018,8,8,16,58,00)
2018-08-08T16:58:00

julia> t2 = DateTime(2021,6,23,10,00,00)
2021-06-23T10:00:00

julia> canonicalize(t2-t1) # creates a CompoundPeriod
149 weeks, 6 days, 17 hours, 2 minutes
```

69.8 Rounding

`Date` and `DateTime` values can be rounded to a specified resolution (e.g., 1 month or 15 minutes) with `floor`, `ceil`, or `round`:

```
julia> floor(Date(1985, 8, 16), Dates.Month)
1985-08-01

julia> ceil(DateTime(2013, 2, 13, 0, 31, 20), Dates.Minute(15))
2013-02-13T00:45:00

julia> round(DateTime(2016, 8, 6, 20, 15), Dates.Day)
2016-08-07T00:00:00
```

Unlike the numeric `round` method, which breaks ties toward the even number by default, the `TimeTyperound` method uses the `RoundNearestTiesUp` rounding mode. (It's difficult to guess what breaking ties to nearest "even" `TimeType` would entail.) Further details on the available `RoundingMode` s can be found in the API reference.

Rounding should generally behave as expected, but there are a few cases in which the expected behaviour is not obvious.

Rounding Epoch

In many cases, the resolution specified for rounding (e.g., `Dates.Second(30)`) divides evenly into the next largest period (in this case, `Dates.Minute(1)`). But rounding behaviour in cases in which this is not true may lead to confusion. What is the expected result of rounding a `DateTime` to the nearest 10 hours?

```
julia> round(DateTime(2016, 7, 17, 11, 55), Dates.Hour(10))
2016-07-17T12:00:00
```

That may seem confusing, given that the hour (12) is not divisible by 10. The reason that `2016-07-17T12:00:00` was chosen is that it is 17,676,660 hours after `0000-01-01T00:00:00`, and 17,676,660 is divisible by 10.

As Julia `Date` and `DateTime` values are represented according to the ISO 8601 standard, `0000-01-01T00:00:00` was chosen as base (or "rounding epoch") from which to begin the count of days (and milliseconds) used in rounding calculations. (Note that this differs slightly from Julia's internal representation of `Date` s using Rata Die notation; but since the ISO 8601 standard is most visible to the end user, `0000-01-01T00:00:00` was chosen as the rounding epoch instead of the `0000-12-31T00:00:00` used internally to minimize confusion.)

The only exception to the use of `0000-01-01T00:00:00` as the rounding epoch is when rounding to weeks. Rounding to the nearest week will always return a Monday (the first day of the week as specified by ISO 8601). For this reason, we use `0000-01-03T00:00:00` (the first day of the first week of year 0000, as defined by ISO 8601) as the base when rounding to a number of weeks.

Here is a related case in which the expected behaviour is not necessarily obvious: What happens when we round to the nearest `P(2)`, where `P` is a `Period` type? In some cases (specifically, when `P <: Dates.TimePeriod`) the answer is clear:

```
julia> round(DateTime(2016, 7, 17, 8, 55, 30), Dates.Hour(2))
2016-07-17T08:00:00

julia> round(DateTime(2016, 7, 17, 8, 55, 30), Dates.Minute(2))
2016-07-17T08:56:00
```

This seems obvious, because two of each of these periods still divides evenly into the next larger order period. But in the case of two months (which still divides evenly into one year), the answer may be surprising:

```
julia> round(DateTime(2016, 7, 17, 8, 55, 30), Dates.Month(2))
2016-07-01T00:00:00
```

Why round to the first day in July, even though it is month 7 (an odd number)? The key is that months are 1-indexed (the first month is assigned 1), unlike hours, minutes, seconds, and milliseconds (the first of which are assigned 0).

This means that rounding a `DateTime` to an even multiple of seconds, minutes, hours, or years (because the ISO 8601 specification includes a year zero) will result in a `DateTime` with an even value in that field, while rounding a `DateTime` to an even multiple of months will result in the months field having an odd value. Because both months and years may contain an irregular number of days, whether rounding to an even number of days will result in an even value in the days field is uncertain.

See the API reference for additional information on methods exported from the `Dates` module.

69.9 API reference

Dates and Time Types

`Dates.Period` – Type.

```
Period
Year
Quarter
Month
Week
Day
Hour
Minute
Second
Millisecond
Microsecond
Nanosecond
```

`Period` types represent discrete, human representations of time.

`Dates.CompoundPeriod` – Type.

```
CompoundPeriod
```

A `CompoundPeriod` is useful for expressing time periods that are not a fixed multiple of smaller periods. For example, "a year and a day" is not a fixed number of days, but can be expressed using a `CompoundPeriod`. In fact, a `CompoundPeriod` is automatically generated by addition of different period types, e.g. `Year(1) + Day(1)` produces a `CompoundPeriod` result.

`Dates.Instant` – Type.

```
Instant
```

`Instant` types represent integer-based, machine representations of time as continuous timelines starting from an epoch.

`Dates.UTInstant` – Type.

```
UTInstant{T}
```

The `UTInstant` represents a machine timeline based on UT time (1 day = one revolution of the earth). The T is a `Period` parameter that indicates the resolution or precision of the instant.

`Dates.TimeType` – Type.

```
TimeType
```

`TimeType` types wrap `Instant` machine instances to provide human representations of the machine instant. `Time`, `DateTime` and `Date` are subtypes of `TimeType`.

`Dates.DateTime` – Type.

```
DateTime
```

`DateTime` represents a point in time according to the proleptic Gregorian calendar. The finest resolution of the time is millisecond (i.e., microseconds or nanoseconds cannot be represented by this type). The type supports fixed-point arithmetic, and thus is prone to underflowing (and overflowing). A notable consequence is rounding when adding a `Microsecond` or a `Nanosecond`:

```
julia> dt = DateTime(2023, 8, 19, 17, 45, 32, 900)
2023-08-19T17:45:32.900

julia> dt + Millisecond(1)
2023-08-19T17:45:32.901

julia> dt + Microsecond(1000) # 1000us == 1ms
2023-08-19T17:45:32.901

julia> dt + Microsecond(999) # 999us rounded to 1000us
2023-08-19T17:45:32.901

julia> dt + Microsecond(1499) # 1499 rounded to 1000us
2023-08-19T17:45:32.901
```

`Dates.Date` – Type.

```
Date
```

`Date` wraps a `UTInstant{Day}` and interprets it according to the proleptic Gregorian calendar.

`Dates.Time` – Type.

```
Time
```

`Time` wraps a `Nanosecond` and represents a specific moment in a 24-hour day.

`Dates.TimeZone` – Type.

```
TimeZone
```

Geographic zone generally based on longitude determining what the time is at a certain location. Some time zones observe daylight savings (eg EST -> EDT). For implementations and more support, see the `TimeZones.jl` package

`Dates.UTC` – Type.

```
UTC
```

`UTC`, or Coordinated Universal Time, is the `TimeZone` from which all others are measured. It is associated with the time at 0° longitude. It is not adjusted for daylight savings.

Dates Functions

`Dates.DateTime` – Method.

```
DateTime(y, [m, d, h, mi, s, ms])::DateTime
```

Construct a `DateTime` type by parts. Arguments must be convertible to `Int64`.

`Dates.DateTime` – Method.

```
DateTime(periods::Period...)::DateTime
```

Construct a `DateTime` type by `Period` type parts. Arguments may be in any order. DateTime parts not provided will default to the value of `Dates.default(period)`.

`Dates.DateTime` – Method.

```
DateTime(f::Function, y[, m, d, h, mi, s]; step=Day(1), limit=10000)::DateTime
```

Create a `DateTime` through the adjuster API. The starting point will be constructed from the provided `y, m, d...` arguments, and will be adjusted until `f::Function` returns `true`. The step size in adjusting can be provided manually through the `step` keyword. `limit` provides a limit to the max number of iterations the adjustment API will pursue before throwing an error (in the case that `f::Function` is never satisfied).

Examples

```
julia> DateTime(dt -> second(dt) == 40, 2010, 10, 20, 10; step = Second(1))
2010-10-20T10:00:40

julia> DateTime(dt -> hour(dt) == 20, 2010, 10, 20, 10; step = Hour(1), limit = 5)
ERROR: ArgumentError: Adjustment limit reached: 5 iterations
Stacktrace:
[...]
```

`Dates.DateTime` – Method.

```
DateTime(dt::Date)
```

Convert a `Date` to a `DateTime`. The hour, minute, second, and millisecond parts of the new `DateTime` are assumed to be zero.

`Dates.DateTime` – Method.

```
DateTime(dt::AbstractString, format::AbstractString; locale="english")
```

Construct a `DateTime` by parsing the `dt` date time string following the pattern given in the `format` string (see `DateFormat` for syntax).

Note

This method creates a `DateFormat` object each time it is called. It is recommended that you create a `DateFormat` object instead and use that as the second argument to avoid performance loss when using the same format repeatedly.

Examples

```
julia> DateTime("2020-01-01", "yyyy-mm-dd")
2020-01-01T00:00:00

julia> a = ("2020-01-01", "2020-01-02");

julia> [DateTime(d, dateformat"yyyy-mm-dd") for d ∈ a] # preferred
2-element Vector{DateTime}:
 2020-01-01T00:00:00
 2020-01-02T00:00:00
```

`Dates.format` – Method.

```
format(dt::TimeType, format::AbstractString; locale="english")::AbstractString
```

Construct a string by using a `TimeType` object and applying the provided `format`. The following character codes can be used to construct the `format` string:

Code	Examples	Comment
y	6	Numeric year with a fixed width
Y	1996	Numeric year with a minimum width
m	1, 12	Numeric month with a minimum width
u	Jan	Month name shortened to 3-chars according to the `locale`
U	January	Full month name according to the `locale` keyword
d	1, 31	Day of the month with a minimum width
H	0, 23	Hour (24-hour clock) with a minimum width
M	0, 59	Minute with a minimum width
S	0, 59	Second with a minimum width
s	000, 500	Millisecond with a minimum width of 3
e	Mon, Tue	Abbreviated days of the week
E	Monday	Full day of week name

The number of sequential code characters indicate the width of the code. A format of `yyyy-mm` specifies that the code y should have a width of four while `m` a width of two. Codes that yield numeric digits have an associated mode: fixed-width or minimum-width. The fixed-width mode left-pads the value with zeros when it is shorter than the specified width and truncates the value when longer. Minimum-width mode works the same as fixed-width except that it does not truncate values longer than the width.

When creating a `format` you can use any non-code characters as a separator. For example to generate the string "1996-01-15T00:00:00" you could use `format`: "yyyy-mm-ddTHH:MM:SS". Note that if you need to use a code character as a literal you can use the escape character backslash. The string "1996y01m" can be produced with the format raw"yyyy\ymm\m".

`Dates.DateFormat` – Type.

```
DateFormat(format::AbstractString, locale="english")
```

Construct a date formatting object that can be used for parsing date strings or formatting a date object as a string. The following character codes can be used to construct the `format` string:

Characters not listed above are normally treated as delimiters between date and time slots. For example a `dt` string of "1996-01-15T00:00:00.0" would have a `format` string like "y-m-dTH:M:S.s". If you

Code	Matches	Comment
Y	1996, 96	Returns year of 1996, 0096
y	1996, 96	Same as Y on parse but discards excess digits on format
m	1, 01	Matches 1 or 2-digit months
u	Jan	Matches abbreviated months according to the locale keyword
U	January	Matches full month names according to the locale keyword
d	1, 01	Matches 1 or 2-digit days
H	00	Matches hours (24-hour clock)
I	00	For outputting hours with 12-hour clock
M	00	Matches minutes
S	00	Matches seconds
s	.500	Matches milliseconds
e	Mon, Tues	Matches abbreviated days of the week
E	Monday	Matches full name days of the week
p	AM	Matches AM/PM (case-insensitive)
yyyymmdd	19960101	Matches fixed-width year, month, and day

need to use a code character as a delimiter you can escape it using backslash. The date "1995y01m" would have the format "y\ym\m".

Note that 12:00AM corresponds 00:00 (midnight), and 12:00PM corresponds to 12:00 (noon). When parsing a time with a p specifier, any hour (either H or I) is interpreted as as a 12-hour clock, so the I code is mainly useful for output.

Creating a DateFormat object is expensive. Whenever possible, create it once and use it many times or try the dateformat"" string macro. Using this macro creates the DateFormat object once at macro expansion time and reuses it later. There are also several pre-defined formatters, listed later.

See DateTime and format for how to use a DateFormat object to parse and write Date strings respectively.

Dates.@dateformat_str – Macro.

```
dateformat"Y-m-d H:M:S"
```

Create a DateFormat object. Similar to DateFormat("Y-m-d H:M:S") but creates the DateFormat object once during macro expansion.

See DateFormat for details about format specifiers.

Dates.DateTime – Method.

```
DateTime(dt::AbstractString, df::DateFormat=ISODateTimeFormat)
```

Construct a DateTime by parsing the dt date time string following the pattern given in the DateFormat object, or dateformat"yyyy-mm-dd\THH:MM:SS.s" if omitted.

Similar to DateTime(::AbstractString, ::AbstractString) but more efficient when repeatedly parsing similarly formatted date time strings with a pre-created DateFormat object.

Dates.Date – Method.

```
Date(y, [m, d])::Date
```

Construct a Date type by parts. Arguments must be convertible to Int64.

Dates.Date – Method.

```
Date(period::Period...)::Date
```

Construct a Date type by Period type parts. Arguments may be in any order. Date parts not provided will default to the value of Dates.default(period).

Dates.Date – Method.

```
Date(f::Function, y[, m, d]; step=Day(1), limit=10000)::Date
```

Create a Date through the adjuster API. The starting point will be constructed from the provided y, m, d arguments, and will be adjusted until f::Function returns true. The step size in adjusting can be provided manually through the step keyword. limit provides a limit to the max number of iterations the adjustment API will pursue before throwing an error (given that f::Function is never satisfied).

Examples

```
julia> Date(date -> week(date) == 20, 2010, 01, 01)
2010-05-17

julia> Date(date -> year(date) == 2010, 2000, 01, 01)
2010-01-01

julia> Date(date -> month(date) == 10, 2000, 01, 01; limit = 5)
ERROR: ArgumentError: Adjustment limit reached: 5 iterations
Stacktrace:
[...]
```

Dates.Date – Method.

```
Date(dt::DateTime)
```

Convert a DateTime to a Date. The hour, minute, second, and millisecond parts of the DateTime are truncated, so only the year, month and day parts are used in construction.

Dates.Date – Method.

```
Date(d::AbstractString, format::AbstractString; locale="english")
```

Construct a Date by parsing the d date string following the pattern given in the format string (see DateFormat for syntax).

Note

This method creates a DateFormat object each time it is called. It is recommended that you create a DateFormat object instead and use that as the second argument to avoid performance loss when using the same format repeatedly.

Examples

```
julia> Date("2020-01-01", "yyyy-mm-dd")
2020-01-01

julia> a = ("2020-01-01", "2020-01-02");

julia> [Date(d, dateformat"yyyy-mm-dd") for d ∈ a] # preferred
2-element Vector{Date}:
 2020-01-01
 2020-01-02
```

`Dates.Date` – Method.

```
Date(d::AbstractString, df::DateFormat=ISODateFormat)
```

Construct a `Date` by parsing the `d` date string following the pattern given in the `DateFormat` object, or dateformat"yyyy-mm-dd" if omitted.

Similar to `Date(::AbstractString, ::AbstractString)` but more efficient when repeatedly parsing similarly formatted date strings with a pre-created `DateFormat` object.

`Dates.Time` – Method.

```
Time(h, [mi, s, ms, us, ns])::Time
```

Construct a `Time` type by parts. Arguments must be convertible to `Int64`.

`Dates.Time` – Method.

```
Time(period::TimePeriod...)::Time
```

Construct a `Time` type by `Period` type parts. Arguments may be in any order. `Time` parts not provided will default to the value of `Dates.default(period)`.

`Dates.Time` – Method.

```
Time(f::Function, h, mi=0; step::Period=Second(1), limit::Int=10000)
Time(f::Function, h, mi, s; step::Period=Millisecond(1), limit::Int=10000)
Time(f::Function, h, mi, s, ms; step::Period=Microsecond(1), limit::Int=10000)
Time(f::Function, h, mi, s, ms, us; step::Period=Nanosecond(1), limit::Int=10000)
```

Create a `Time` through the adjuster API. The starting point will be constructed from the provided `h, mi, s, ms, us` arguments, and will be adjusted until `f::Function` returns `true`. The step size in adjusting can be provided manually through the `step` keyword. `limit` provides a limit to the max number of iterations the adjustment API will pursue before throwing an error (in the case that `f::Function` is never satisfied). Note that the default step will adjust to allow for greater precision for the given arguments; i.e. if hour, minute, and second arguments are provided, the default step will be `Millisecond(1)` instead of `Second(1)`.

Examples

```
julia> Time(t -> minute(t) == 30, 20)
20:30:00

julia> Time(t -> minute(t) == 0, 20)
```

```
20:00:00

julia> Time(t -> hour(t) == 10, 3; limit = 5)
ERROR: ArgumentError: Adjustment limit reached: 5 iterations
Stacktrace:
[...]
```

`Dates.Time` – Method.

```
Time(dt::DateTime)
```

Convert a `DateTime` to a `Time`. The hour, minute, second, and millisecond parts of the `DateTime` are used to create the new `Time`. Microsecond and nanoseconds are zero by default.

`Dates.Time` – Method.

```
Time(t::AbstractString, format::AbstractString; locale="english")
```

Construct a `Time` by parsing the `t` time string following the pattern given in the `format` string (see `DateFormat` for syntax).

Note

This method creates a `DateFormat` object each time it is called. It is recommended that you create a `DateFormat` object instead and use that as the second argument to avoid performance loss when using the same format repeatedly.

Examples

```
julia> Time("12:34pm", "HH:MMp")
12:34:00

julia> a = ("12:34pm", "2:34am");

julia> [Time(d, dateformat"HH:MMp") for d ∈ a] # preferred
2-element Vector{Time}:
 12:34:00
 02:34:00
```

`Dates.Time` – Method.

```
Time(t::AbstractString, df::DateFormat=ISOTimeFormat)
```

Construct a `Time` by parsing the `t` date time string following the pattern given in the `DateFormat` object, or dateformat"HH:MM:SS.s" if omitted.

Similar to `Time(::AbstractString, ::AbstractString)` but more efficient when repeatedly parsing similarly formatted time strings with a pre-created `DateFormat` object.

`Dates.now` – Method.

```
now()::DateTime
```

Return a `DateTime` corresponding to the user's system time including the system timezone locale.

Dates.now – Method.

```
now(::Type{UTC})::DateTime
```

Return a DateTime corresponding to the user's system time as UTC/GMT. For other time zones, see the TimeZones.jl package.

Examples

```
julia> now(UTC)
2023-01-04T10:52:24.864
```

Base.eps – Method.

```
eps(::Type{DateTime})::Millisecond
eps(::Type{Date})::Day
eps(::Type{Time})::Nanosecond
eps(::TimeType)::Period
```

Return the smallest unit value supported by the TimeType.

Examples

```
julia> eps(DateTime)
1 millisecond

julia> eps(Date)
1 day

julia> eps(Time)
1 nanosecond
```

Accessor Functions

Dates.year – Function.

```
year(dt::TimeType)::Int64
```

The year of a Date or DateTime as an Int64.

Dates.month – Function.

```
month(dt::TimeType)::Int64
```

The month of a Date or DateTime as an Int64.

Dates.week – Function.

```
week(dt::TimeType)::Int64
```

Return the ISO week date of a Date or DateTime as an Int64. Note that the first week of a year is the week that contains the first Thursday of the year, which can result in dates prior to January 4th being in the last week of the previous year. For example, week(Date(2005, 1, 1)) is the 53rd week of 2004.

Examples

```
julia> week(Date(1989, 6, 22))
25

julia> week(Date(2005, 1, 1))
53

julia> week(Date(2004, 12, 31))
53
```

`Dates.day` – Function.

```
day(dt::TimeType)::Int64
```

The day of month of a `Date` or `DateTime` as an `Int64`.

`Dates.hour` – Function.

```
hour(dt::DateTime)::Int64
```

The hour of day of a `DateTime` as an `Int64`.

```
hour(t::Time)::Int64
```

The hour of a `Time` as an `Int64`.

`Dates.minute` – Function.

```
minute(dt::DateTime)::Int64
```

The minute of a `DateTime` as an `Int64`.

```
minute(t::Time)::Int64
```

The minute of a `Time` as an `Int64`.

`Dates.second` – Function.

```
second(dt::DateTime)::Int64
```

The second of a `DateTime` as an `Int64`.

```
second(t::Time)::Int64
```

The second of a `Time` as an `Int64`.

`Dates.millisecond` – Function.

```
millisecond(dt::DateTime)::Int64
```

The millisecond of a `DateTime` as an `Int64`.

```
millisecond(t::Time)::Int64
```

The millisecond of a `Time` as an `Int64`.

Dates.microsecond – Function.

```
microsecond(t::Time)::Int64
```

The microsecond of a Time as an Int64.

Dates.nanosecond – Function.

```
nanosecond(t::Time)::Int64
```

The nanosecond of a Time as an Int64.

Dates.Year – Method.

```
Year(v)
```

Construct a Year object with the given v value. Input must be losslessly convertible to an Int64.

Dates.Month – Method.

```
Month(v)
```

Construct a Month object with the given v value. Input must be losslessly convertible to an Int64.

Dates.Week – Method.

```
Week(v)
```

Construct a Week object with the given v value. Input must be losslessly convertible to an Int64.

Dates.Day – Method.

```
Day(v)
```

Construct a Day object with the given v value. Input must be losslessly convertible to an Int64.

Dates.Hour – Method.

```
Hour(dt::DateTime)
```

The hour part of a DateTime as a Hour.

Dates.Minute – Method.

```
Minute(dt::DateTime)
```

The minute part of a DateTime as a Minute.

Dates.Second – Method.

```
Second(dt::DateTime)
```

The second part of a DateTime as a Second.

Dates.Millisecond – Method.

```
Millisecond(dt::DateTime)
```

The millisecond part of a DateTime as a `Millisecond`.

`Dates.Microsecond` – Method.

```
Microsecond(dt::Time)
```

The microsecond part of a Time as a `Microsecond`.

`Dates.Nanosecond` – Method.

```
Nanosecond(dt::Time)
```

The nanosecond part of a Time as a `Nanosecond`.

`Dates.yearmonth` – Function.

```
yearmonth(dt::TimeType) -> (Int64, Int64)
```

Simultaneously return the year and month parts of a `Date` or `DateTime`.

`Dates.monthday` – Function.

```
monthday(dt::TimeType) -> (Int64, Int64)
```

Simultaneously return the month and day parts of a `Date` or `DateTime`.

`Dates.yearmonthday` – Function.

```
yearmonthday(dt::TimeType) -> (Int64, Int64, Int64)
```

Simultaneously return the year, month and day parts of a `Date` or `DateTime`.

Query Functions

`Dates.dayname` – Function.

```
dayname(dt::TimeType; locale="english")::String
dayname(day::Integer; locale="english")::String
```

Return the full day name corresponding to the day of the week of the `Date` or `DateTime` in the given `locale`. Also accepts `Integer`.

Examples

```
julia> dayname(Date("2000-01-01"))
"Saturday"

julia> dayname(4)
"Thursday"
```

`Dates.dayabbr` – Function.

```
dayabbr(dt::TimeType; locale="english")::String
dayabbr(day::Integer; locale="english")::String
```

Return the abbreviated name corresponding to the day of the week of the `Date` or `DateTime` in the given `locale`. Also accepts `Integer`.

Examples

```
julia> dayabbr(Date("2000-01-01"))
"Sat"

julia> dayabbr(3)
"Wed"
```

`Dates.dayofweek` – Function.

```
dayofweek(dt::TimeType)::Int64
```

Return the day of the week as an `Int64` with `1 = Monday, 2 = Tuesday, etc.`.

Examples

```
julia> dayofweek(Date("2000-01-01"))
6
```

`Dates.dayofmonth` – Function.

```
dayofmonth(dt::TimeType)::Int64
```

The day of month of a `Date` or `DateTime` as an `Int64`.

`Dates.dayofweekofmonth` – Function.

```
dayofweekofmonth(dt::TimeType)::Int
```

For the day of week of `dt`, return which number it is in `dt`'s month. So if the day of the week of `dt` is Monday, then `1 = First Monday of the month, 2 = Second Monday of the month, etc.` In the range 1:5.

Examples

```
julia> dayofweekofmonth(Date("2000-02-01"))
1

julia> dayofweekofmonth(Date("2000-02-08"))
2

julia> dayofweekofmonth(Date("2000-02-15"))
3
```

`Dates.daysofweekinmonth` – Function.

```
daysofweekinmonth(dt::TimeType)::Int
```

For the day of week of `dt`, return the total number of that day of the week in `dt`'s month. Returns 4 or 5. Useful in temporal expressions for specifying the last day of a week in a month by including `dayofweekofmonth(dt) == daysofweekinmonth(dt)` in the adjuster function.

Examples

```
julia> daysofweekinmonth(Date("2005-01-01"))
5

julia> daysofweekinmonth(Date("2005-01-04"))
4
```

`Dates.monthname` – Function.

```
monthname(dt::TimeType; locale="english")::String
monthname(month::Integer, locale="english")::String
```

Return the full name of the month of the `Date` or `DateTime` or `Integer` in the given `locale`.

Examples

```
julia> monthname(Date("2005-01-04"))
"January"

julia> monthname(2)
"February"
```

`Dates.monthabbr` – Function.

```
monthabbr(dt::TimeType; locale="english")::String
monthabbr(month::Integer, locale="english")::String
```

Return the abbreviated month name of the `Date` or `DateTime` or `Integer` in the given `locale`.

Examples

```
julia> monthabbr(Date("2005-01-04"))
"Jan"

julia> monthabbr(2)
"Feb"
```

`Dates.daysinmonth` – Function.

```
daysinmonth(dt::TimeType)::Int
```

Return the number of days in the month of `dt`. Value will be 28, 29, 30, or 31.

Examples

```
julia> daysinmonth(Date("2000-01"))
31

julia> daysinmonth(Date("2001-02"))
28

julia> daysinmonth(Date("2000-02"))
29
```

`Dates.isleapyear` – Function.

```
isleapyear(dt::TimeType)::Bool
```

Return `true` if the year of `dt` is a leap year.

Examples

```
julia> isleapyear(Date("2004"))
true

julia> isleapyear(Date("2005"))
false
```

`Dates.dayofyear` – Function.

```
dayofyear(dt::TimeType)::Int
```

Return the day of the year for `dt` with January 1st being day 1.

`Dates.daysinyear` – Function.

```
daysinyear(dt::TimeType)::Int
```

Return 366 if the year of `dt` is a leap year, otherwise return 365.

Examples

```
julia> daysinyear(1999)
365

julia> daysinyear(2000)
366
```

`Dates.quarterofyear` – Function.

```
quarterofyear(dt::TimeType)::Int
```

Return the quarter that `dt` resides in. Range of value is 1:4.

`Dates.dayofquarter` – Function.

```
dayofquarter(dt::TimeType)::Int
```

Return the day of the current quarter of `dt`. Range of value is 1:92.

Adjuster Functions

`Base.trunc` – Method.

```
trunc(dt::TimeType, ::Type{Period})::TimeType
```

Truncates the value of `dt` according to the provided `Period` type.

Examples

```
julia> trunc(DateTime("1996-01-01T12:30:00"), Day)
1996-01-01T00:00:00
```

`Dates.firstdayofweek` – Function.

```
firstdayofweek(dt::TimeType)::TimeType
```

Adjusts `dt` to the Monday of its week.

Examples

```
julia> firstdayofweek(DateTime("1996-01-05T12:30:00"))
1996-01-01T00:00:00
```

`Dates.lastdayofweek` – Function.

```
lastdayofweek(dt::TimeType)::TimeType
```

Adjusts `dt` to the Sunday of its week.

Examples

```
julia> lastdayofweek(DateTime("1996-01-05T12:30:00"))
1996-01-07T00:00:00
```

`Dates.firstdayofmonth` – Function.

```
firstdayofmonth(dt::TimeType)::TimeType
```

Adjusts `dt` to the first day of its month.

Examples

```
julia> firstdayofmonth(DateTime("1996-05-20"))
1996-05-01T00:00:00
```

`Dates.lastdayofmonth` – Function.

```
lastdayofmonth(dt::TimeType)::TimeType
```

Adjusts `dt` to the last day of its month.

Examples

```
julia> lastdayofmonth(DateTime("1996-05-20"))
1996-05-31T00:00:00
```

`Dates.firstdayofyear` – Function.

```
firstdayofyear(dt::TimeType)::TimeType
```

Adjusts `dt` to the first day of its year.

Examples

```
julia> firstdayofyear(DateTime("1996-05-20"))
1996-01-01T00:00:00
```

`Dates.lastdayofyear` – Function.

```
lastdayofyear(dt::TimeType)::TimeType
```

Adjusts `dt` to the last day of its year.

Examples

```
julia> lastdayofyear(DateTime("1996-05-20"))
1996-12-31T00:00:00
```

`Dates.firstdayofquarter` – Function.

```
firstdayofquarter(dt::TimeType)::TimeType
```

Adjusts `dt` to the first day of its quarter.

Examples

```
julia> firstdayofquarter(DateTime("1996-05-20"))
1996-04-01T00:00:00

julia> firstdayofquarter(DateTime("1996-08-20"))
1996-07-01T00:00:00
```

`Dates.lastdayofquarter` – Function.

```
lastdayofquarter(dt::TimeType)::TimeType
```

Adjusts `dt` to the last day of its quarter.

Examples

```
julia> lastdayofquarter(DateTime("1996-05-20"))
1996-06-30T00:00:00

julia> lastdayofquarter(DateTime("1996-08-20"))
1996-09-30T00:00:00
```

`Dates.tonext` – Method.

```
tonext(dt::TimeType, dow::Int; same::Bool=false)::TimeType
```

Adjusts `dt` to the next day of week corresponding to `dow` with `1 = Monday, 2 = Tuesday, etc`. Setting `same=true` allows the current `dt` to be considered as the next `dow`, allowing for no adjustment to occur.

`Dates.toprev` – Method.

```
toprev(dt::TimeType, dow::Int; same::Bool=false)::TimeType
```

Adjusts dt to the previous day of week corresponding to dow with 1 = Monday, 2 = Tuesday, etc. Setting same=true allows the current dt to be considered as the previous dow, allowing for no adjustment to occur.

Dates.tofirst – Function.

```
tofirst(dt::TimeType, dow::Int; of=Month)::TimeType
```

Adjusts dt to the first dow of its month. Alternatively, of=Year will adjust to the first dow of the year.

Dates.tolast – Function.

```
tolast(dt::TimeType, dow::Int; of=Month)::TimeType
```

Adjusts dt to the last dow of its month. Alternatively, of=Year will adjust to the last dow of the year.

Dates.tonext – Method.

```
tonext(func::Function, dt::TimeType; step=Day(1), limit=10000, same=false)::TimeType
```

Adjusts dt by iterating at most limit iterations by step increments until func returns true. func must take a single TimeType argument and return a Bool. same allows dt to be considered in satisfying func.

Dates.toprev – Method.

```
toprev(func::Function, dt::TimeType; step=Day(-1), limit=10000, same=false)::TimeType
```

Adjusts dt by iterating at most limit iterations by step increments until func returns true. func must take a single TimeType argument and return a Bool. same allows dt to be considered in satisfying func.

Periods

Dates.Period – Method.

```
Year(v)
Quarter(v)
Month(v)
Week(v)
Day(v)
Hour(v)
Minute(v)
Second(v)
Millisecond(v)
Microsecond(v)
Nanosecond(v)
```

Construct a Period type with the given v value. Input must be losslessly convertible to an Int64.

Dates.CompoundPeriod – Method.

```
CompoundPeriod(periods)
```

Construct a CompoundPeriod from a Vector of Periods. All Periods of the same type will be added together.

Examples

```
julia> Dates.CompoundPeriod(Dates.Hour(12), Dates.Hour(13))
25 hours

julia> Dates.CompoundPeriod(Dates.Hour(-1), Dates.Minute(1))
-1 hour, 1 minute

julia> Dates.CompoundPeriod(Dates.Month(1), Dates.Week(-2))
1 month, -2 weeks

julia> Dates.CompoundPeriod(Dates.Minute(50000))
50000 minutes
```

`Dates.canonicalize` – Function.

```
canonicalize(::CompoundPeriod)::CompoundPeriod
```

Reduces the `CompoundPeriod` into its canonical form by applying the following rules:

- Any `Period` large enough be partially representable by a coarser `Period` will be broken into multiple `Periods` (eg. `Hour(30)` becomes `Day(1) + Hour(6)`)
- `Periods` with opposite signs will be combined when possible (eg. `Hour(1) - Day(1)` becomes `-Hour(23)`)

Examples

```
julia> canonicalize(Dates.CompoundPeriod(Dates.Hour(12), Dates.Hour(13)))
1 day, 1 hour

julia> canonicalize(Dates.CompoundPeriod(Dates.Hour(-1), Dates.Minute(1)))
-59 minutes

julia> canonicalize(Dates.CompoundPeriod(Dates.Month(1), Dates.Week(-2)))
1 month, -2 weeks

julia> canonicalize(Dates.CompoundPeriod(Dates.Minute(50000)))
4 weeks, 6 days, 17 hours, 20 minutes
```

`Dates.value` – Function.

```
Dates.value(x::Period)::Int64
```

For a given period, return the value associated with that period. For example, `value(Millisecond(10))` returns 10 as an integer.

`Dates.default` – Function.

```
default(p::Period)::Period
```

Return a sensible "default" value for the input Period by returning `T(1)` for Year, Month, and Day, and `T(0)` for Hour, Minute, Second, and Millisecond.

`Dates.periods` – Function.

```
Dates.periods(::CompoundPeriod)::Vector{Period}
```

Return the `Vector` of `Periods` that comprise the given `CompoundPeriod`.

Julia 1.7

This function requires Julia 1.7 or later.

Rounding Functions

`Date` and `DateTime` values can be rounded to a specified resolution (e.g., 1 month or 15 minutes) with `floor`, `ceil`, or `round`.

`Base.floor` – Method.

```
floor(dt::TimeType, p::Period)::TimeType
```

Return the nearest `Date` or `DateTime` less than or equal to `dt` at resolution `p`.

For convenience, `p` may be a type instead of a value: `floor(dt, Dates.Hour)` is a shortcut for `floor(dt, Dates.Hour(1))`.

```
julia> floor(Date(1985, 8, 16), Month)
1985-08-01

julia> floor(DateTime(2013, 2, 13, 0, 31, 20), Minute(15))
2013-02-13T00:30:00

julia> floor(DateTime(2016, 8, 6, 12, 0, 0), Day)
2016-08-06T00:00:00
```

`Base.ceil` – Method.

```
ceil(dt::TimeType, p::Period)::TimeType
```

Return the nearest `Date` or `DateTime` greater than or equal to `dt` at resolution `p`.

For convenience, `p` may be a type instead of a value: `ceil(dt, Dates.Hour)` is a shortcut for `ceil(dt, Dates.Hour(1))`.

```
julia> ceil(Date(1985, 8, 16), Month)
1985-09-01

julia> ceil(DateTime(2013, 2, 13, 0, 31, 20), Minute(15))
2013-02-13T00:45:00

julia> ceil(DateTime(2016, 8, 6, 12, 0, 0), Day)
2016-08-07T00:00:00
```

`Base.round` – Method.

```
round(dt::TimeType, p::Period, [r::RoundingMode]) -> TimeType
```

Return the `Date` or `DateTime` nearest to `dt` at resolution `p`. By default (`RoundNearestTiesUp`), ties (e.g., rounding 9:30 to the nearest hour) will be rounded up.

For convenience, `p` may be a type instead of a value: `round(dt, Dates.Hour)` is a shortcut for `round(dt, Dates.Hour(1))`.

```
julia> round(Date(1985, 8, 16), Month)
1985-08-01

julia> round(DateTime(2013, 2, 13, 0, 31, 20), Minute(15))
2013-02-13T00:30:00

julia> round(DateTime(2016, 8, 6, 12, 0, 0), Day)
2016-08-07T00:00:00
```

Valid rounding modes for `round(::TimeType, ::Period, ::RoundingMode)` are `RoundNearestTiesUp` (default), `RoundDown` (`floor`), and `RoundUp` (`ceil`).

Most `Period` values can also be rounded to a specified resolution:

`Base.floor` – Method.

```
floor(x::Period, precision::T) where T <: Union{TimePeriod, Week, Day} -> T
```

Round `x` down to the nearest multiple of `precision`. If `x` and `precision` are different subtypes of `Period`, the return value will have the same type as `precision`.

For convenience, `precision` may be a type instead of a value: `floor(x, Dates.Hour)` is a shortcut for `floor(x, Dates.Hour(1))`.

```
julia> floor(Day(16), Week)
2 weeks

julia> floor(Minute(44), Minute(15))
30 minutes

julia> floor(Hour(36), Day)
1 day
```

Rounding to a `precision` of `Months` or `Years` is not supported, as these `Periods` are of inconsistent length.

`Base.ceil` – Method.

```
ceil(x::Period, precision::T) where T <: Union{TimePeriod, Week, Day} -> T
```

Round `x` up to the nearest multiple of `precision`. If `x` and `precision` are different subtypes of `Period`, the return value will have the same type as `precision`.

For convenience, `precision` may be a type instead of a value: `ceil(x, Dates.Hour)` is a shortcut for `ceil(x, Dates.Hour(1))`.

```
julia> ceil(Day(16), Week)
3 weeks
```

```
julia> ceil(Minute(44), Minute(15))
45 minutes

julia> ceil(Hour(36), Day)
2 days
```

Rounding to a `precision` of `Month`s or `Year`s is not supported, as these `Period`s are of inconsistent length.

`Base.round` – Method.

```
round(x::Period, precision::T, [r::RoundingMode]) where T <: Union{TimePeriod, Week, Day} -> T
```

Round `x` to the nearest multiple of `precision`. If `x` and `precision` are different subtypes of `Period`, the return value will have the same type as `precision`. By default (`RoundNearestTiesUp`), ties (e.g., rounding 90 minutes to the nearest hour) will be rounded up.

For convenience, `precision` may be a type instead of a value: `round(x, Dates.Hour)` is a shortcut for `round(x, Dates.Hour(1))`.

```
julia> round(Day(16), Week)
2 weeks

julia> round(Minute(44), Minute(15))
45 minutes

julia> round(Hour(36), Day)
2 days
```

Valid rounding modes for `round(::Period, ::T, ::RoundingMode)` are `RoundNearestTiesUp` (default), `RoundDown` (`floor`), and `RoundUp` (`ceil`).

Rounding to a `precision` of `Month`s or `Year`s is not supported, as these `Period`s are of inconsistent length.

The following functions are not exported:

`Dates.floorceil` – Function.

```
floorceil(dt::TimeType, p::Period) -> (TimeType, TimeType)
```

Simultaneously return the `floor` and `ceil` of a `Date` or `DateTime` at resolution `p`. More efficient than calling both `floor` and `ceil` individually.

```
floorceil(x::Period, precision::T) where T <: Union{TimePeriod, Week, Day} -> (T, T)
```

Simultaneously return the `floor` and `ceil` of `Period` at resolution `p`. More efficient than calling both `floor` and `ceil` individually.

`Dates.epochdays2date` – Function.

```
epochdays2date(days)::Date
```

Take the number of days since the rounding epoch (`0000-01-01T00:00:00`) and return the corresponding `Date`.

`Dates.epochms2datetime` – Function.

```
epochms2datetime(milliseconds)::DateTime
```

Take the number of milliseconds since the rounding epoch (`0000-01-01T00:00:00`) and return the corresponding `DateTime`.

`Dates.date2epochdays` – Function.

```
date2epochdays(dt::Date)::Int64
```

Take the given `Date` and return the number of days since the rounding epoch (`0000-01-01T00:00:00`) as an `Int64`.

`Dates.datetime2epochms` – Function.

```
datetime2epochms(dt::DateTime)::Int64
```

Take the given `DateTime` and return the number of milliseconds since the rounding epoch (`0000-01-01T00:00:00`) as an `Int64`.

Conversion Functions

`Dates.today` – Function.

```
today()::Date
```

Return the date portion of `now()`.

`Dates.unix2datetime` – Function.

```
unix2datetime(x)::DateTime
```

Take the number of seconds since unix epoch `1970-01-01T00:00:00` and convert to the corresponding `DateTime`.

`Dates.datetime2unix` – Function.

```
datetime2unix(dt::DateTime)::Float64
```

Take the given `DateTime` and return the number of seconds since the unix epoch `1970-01-01T00:00:00` as a `Float64`.

`Dates.julian2datetime` – Function.

```
julian2datetime(julian_days)::DateTime
```

Take the number of Julian calendar days since epoch `-4713-11-24T12:00:00` and return the corresponding `DateTime`.

`Dates.datetime2julian` – Function.

```
datetime2julian(dt::DateTime)::Float64
```

Take the given `DateTime` and return the number of Julian calendar days since the julian epoch `-4713-11-24T12:00:00` as a `Float64`.

`Dates.rata2datetime` – Function.

```
rata2datetime(days)::DateTime
```

Take the number of Rata Die days since epoch `0000-12-31T00:00:00` and return the corresponding `DateTime`.

`Dates.datetime2rata` – Function.

```
datetime2rata(dt::TimeType)::Int64
```

Return the number of Rata Die days since epoch from the given `Date` or `DateTime`.

Constants

Days of the Week:

Variable	Abbr.	Value (Int)
`Monday`	`Mon`	1
`Tuesday`	`Tue`	2
`Wednesday`	`Wed`	3
`Thursday`	`Thu`	4
`Friday`	`Fri`	5
`Saturday`	`Sat`	6
`Sunday`	`Sun`	7

Months of the Year:

Variable	Abbr.	Value (Int)
`January`	`Jan`	1
`February`	`Feb`	2
`March`	`Mar`	3
`April`	`Apr`	4
`May`	`May`	5
`June`	`Jun`	6
`July`	`Jul`	7
`August`	`Aug`	8
`September`	`Sep`	9
`October`	`Oct`	10
`November`	`Nov`	11
`December`	`Dec`	12

Common Date Formatters

`Dates.ISODateTimeFormat` – Constant.

```
Dates.ISODateTimeFormat
```

Describes the ISO8601 formatting for a date and time. This is the default value for `Dates.format` of a `DateTime`.

Examples

```
julia> Dates.format(DateTime(2018, 8, 8, 12, 0, 43, 1), ISODateTimeFormat)
"2018-08-08T12:00:43.001"
```

`Dates.ISODateFormat` – Constant.

```
Dates.ISODateFormat
```

Describes the ISO8601 formatting for a date. This is the default value for `Dates.format` of a `Date`.

Examples

```
julia> Dates.format(Date(2018, 8, 8), ISODateFormat)
"2018-08-08"
```

`Dates.ISOTimeFormat` – Constant.

```
Dates.ISOTimeFormat
```

Describes the ISO8601 formatting for a time. This is the default value for `Dates.format` of a `Time`.

Examples

```
julia> Dates.format(Time(12, 0, 43, 1), ISOTimeFormat)
"12:00:43.001"
```

`Dates.RFC1123Format` – Constant.

```
Dates.RFC1123Format
```

Describes the RFC1123 formatting for a date and time.

Examples

```
julia> Dates.format(DateTime(2018, 8, 8, 12, 0, 43, 1), RFC1123Format)
"Wed, 08 Aug 2018 12:00:43"
```

Chapter 70

Delimited Files

`DelimitedFiles.readdlm` – Method.

```
readdlm(source, delim::AbstractChar, T::Type, eol::AbstractChar; header=false, skipstart=0,
↪  skipblanks=true, use_mmap, quotes=true, dims, comments=false, comment_char='#')
```

Read a matrix from the source where each line (separated by `eol`) gives one row, with elements separated by the given delimiter. The source can be a text file, stream or byte array. Memory mapped files can be used by passing the byte array representation of the mapped segment as source.

If T is a numeric type, the result is an array of that type, with any non-numeric elements as `NaN` for floating-point types, or zero. Other useful values of T include `String`, `AbstractString`, and `Any`.

If `header` is `true`, the first row of data will be read as header and the tuple `(data_cells, header_cells)` is returned instead of only `data_cells`.

Specifying `skipstart` will ignore the corresponding number of initial lines from the input.

If `skipblanks` is `true`, blank lines in the input will be ignored.

If `use_mmap` is `true`, the file specified by `source` is memory mapped for potential speedups if the file is large. Default is `false`. On a Windows filesystem, `use_mmap` should not be set to `true` unless the file is only read once and is also not written to. Some edge cases exist where an OS is Unix-like but the filesystem is Windows-like.

If `quotes` is `true`, columns enclosed within double-quote (") characters are allowed to contain new lines and column delimiters. Double-quote characters within a quoted field must be escaped with another double-quote. Specifying `dims` as a tuple of the expected rows and columns (including header, if any) may speed up reading of large files. If `comments` is `true`, lines beginning with `comment_char` and text following `comment_char` in any line are ignored.

Examples

```
julia> using DelimitedFiles

julia> x = [1; 2; 3; 4];

julia> y = [5; 6; 7; 8];

julia> open("delim_file.txt", "w") do io
```

```
          writedlm(io, [x y])
      end

julia> readdlm("delim_file.txt", '\t', Int, '\n')
4×2 Matrix{Int64}:
 1  5
 2  6
 3  7
 4  8

julia> rm("delim_file.txt")
```

DelimitedFiles.readdlm – Method.

```
readdlm(source, delim::AbstractChar, eol::AbstractChar; options...)
```

If all data is numeric, the result will be a numeric array. If some elements cannot be parsed as numbers, a heterogeneous array of numbers and strings is returned.

DelimitedFiles.readdlm – Method.

```
readdlm(source, delim::AbstractChar, T::Type; options...)
```

The end of line delimiter is taken as \n.

Examples

```
julia> using DelimitedFiles

julia> x = [1; 2; 3; 4];

julia> y = [1.1; 2.2; 3.3; 4.4];

julia> open("delim_file.txt", "w") do io
          writedlm(io, [x y], ',')
      end;

julia> readdlm("delim_file.txt", ',', Float64)
4×2 Matrix{Float64}:
 1.0  1.1
 2.0  2.2
 3.0  3.3
 4.0  4.4

julia> rm("delim_file.txt")
```

DelimitedFiles.readdlm – Method.

```
readdlm(source, delim::AbstractChar; options...)
```

The end of line delimiter is taken as \n. If all data is numeric, the result will be a numeric array. If some elements cannot be parsed as numbers, a heterogeneous array of numbers and strings is returned.

Examples

```
julia> using DelimitedFiles

julia> x = [1; 2; 3; 4];

julia> y = [1.1; 2.2; 3.3; 4.4];

julia> open("delim_file.txt", "w") do io
           writedlm(io, [x y], ',')
       end;

julia> readdlm("delim_file.txt", ',')
4×2 Matrix{Float64}:
 1.0  1.1
 2.0  2.2
 3.0  3.3
 4.0  4.4

julia> z = ["a"; "b"; "c"; "d"];

julia> open("delim_file.txt", "w") do io
           writedlm(io, [x z], ',')
       end;

julia> readdlm("delim_file.txt", ',')
4×2 Matrix{Any}:
 1  "a"
 2  "b"
 3  "c"
 4  "d"

julia> rm("delim_file.txt")
```

`DelimitedFiles.readdlm` – Method.

```
readdlm(source, T::Type; options...)
```

The columns are assumed to be separated by one or more whitespaces. The end of line delimiter is taken as \n.

Examples

```
julia> using DelimitedFiles

julia> x = [1; 2; 3; 4];

julia> y = [5; 6; 7; 8];

julia> open("delim_file.txt", "w") do io
           writedlm(io, [x y])
       end;

julia> readdlm("delim_file.txt", Int64)
4×2 Matrix{Int64}:
 1  5
```

```
 2  6
 3  7
 4  8

julia> readdlm("delim_file.txt", Float64)
4×2 Matrix{Float64}:
 1.0  5.0
 2.0  6.0
 3.0  7.0
 4.0  8.0

julia> rm("delim_file.txt")
```

`DelimitedFiles.readdlm` – Method.

```
readdlm(source; options...)
```

The columns are assumed to be separated by one or more whitespaces. The end of line delimiter is taken as `\n`. If all data is numeric, the result will be a numeric array. If some elements cannot be parsed as numbers, a heterogeneous array of numbers and strings is returned.

Examples

```
julia> using DelimitedFiles

julia> x = [1; 2; 3; 4];

julia> y = ["a"; "b"; "c"; "d"];

julia> open("delim_file.txt", "w") do io
           writedlm(io, [x y])
       end;

julia> readdlm("delim_file.txt")
4×2 Matrix{Any}:
 1  "a"
 2  "b"
 3  "c"
 4  "d"

julia> rm("delim_file.txt")
```

`DelimitedFiles.writedlm` – Function.

```
writedlm(f, A, delim='\t'; opts)
```

Write `A` (a vector, matrix, or an iterable collection of iterable rows) as text to `f` (either a filename string or an `IO` stream) using the given delimiter `delim` (which defaults to tab, but can be any printable Julia object, typically a `Char` or `AbstractString`).

For example, two vectors `x` and `y` of the same length can be written as two columns of tab-delimited text to `f` by either `writedlm(f, [x y])` or by `writedlm(f, zip(x, y))`.

Examples

```
julia> using DelimitedFiles

julia> x = [1; 2; 3; 4];

julia> y = [5; 6; 7; 8];

julia> open("delim_file.txt", "w") do io
           writedlm(io, [x y])
       end

julia> readdlm("delim_file.txt", '\t', Int, '\n')
4×2 Matrix{Int64}:
 1  5
 2  6
 3  7
 4  8

julia> rm("delim_file.txt")
```

Chapter 71

Distributed Computing

`Distributed` – Module.

Tools for distributed parallel processing.

`Distributed.addprocs` – Function.

```
addprocs(manager::ClusterManager; kwargs...) -> List of process identifiers
```

Launches worker processes via the specified cluster manager.

For example, Beowulf clusters are supported via a custom cluster manager implemented in the package `ClusterManagers.jl`.

The number of seconds a newly launched worker waits for connection establishment from the master can be specified via variable `JULIA_WORKER_TIMEOUT` in the worker process's environment. Relevant only when using TCP/IP as transport.

To launch workers without blocking the REPL, or the containing function if launching workers programmatically, execute `addprocs` in its own task.

Examples

```
# On busy clusters, call `addprocs` asynchronously
t = @async addprocs(...)
```

```
# Utilize workers as and when they come online
if nprocs() > 1   # Ensure at least one new worker is available
   ....   # perform distributed execution
end
```

```
# Retrieve newly launched worker IDs, or any error messages
if istaskdone(t)   # Check if `addprocs` has completed to ensure `fetch` doesn't block
    if nworkers() == N
        new_pids = fetch(t)
    else
        fetch(t)
    end
end
```

```
addprocs(machines; tunnel=false, sshflags=``, max_parallel=10, kwargs...) -> List of process
↪  identifiers
```

Add worker processes on remote machines via SSH. Configuration is done with keyword arguments (see below). In particular, the exename keyword can be used to specify the path to the `julia` binary on the remote machine(s).

`machines` is a vector of "machine specifications" which are given as strings of the form `[user@]host[:port] [bind_addr[:port]]`. `user` defaults to current user and `port` to the standard SSH port. If `[bind_addr[:port]]` is specified, other workers will connect to this worker at the specified `bind_addr` and `port`.

It is possible to launch multiple processes on a remote host by using a tuple in the `machines` vector or the form `(machine_spec, count)`, where `count` is the number of workers to be launched on the specified host. Passing `:auto` as the worker count will launch as many workers as the number of CPU threads on the remote host.

Examples:

```
addprocs([
    "remote1",               # one worker on 'remote1' logging in with the current username
    "user@remote2",          # one worker on 'remote2' logging in with the 'user' username
    "user@remote3:2222",     # specifying SSH port to '2222' for 'remote3'
    ("user@remote4", 4),     # launch 4 workers on 'remote4'
    ("user@remote5", :auto), # launch as many workers as CPU threads on 'remote5'
])
```

Keyword arguments:

- `tunnel`: if `true` then SSH tunneling will be used to connect to the worker from the master process. Default is `false`.
- `multiplex`: if `true` then SSH multiplexing is used for SSH tunneling. Default is `false`.
- `ssh`: the name or path of the SSH client executable used to start the workers. Default is `"ssh"`.
- `sshflags`: specifies additional ssh options, e.g. ``sshflags=`-i /home/foo/bar.pem` ``
- `max_parallel`: specifies the maximum number of workers connected to in parallel at a host. Defaults to 10.
- `shell`: specifies the type of shell to which ssh connects on the workers.
  - `shell=:posix`: a POSIX-compatible Unix/Linux shell (sh, ksh, bash, dash, zsh, etc.). The default.
  - `shell=:csh`: a Unix C shell (csh, tcsh).
  - `shell=:wincmd`: Microsoft Windows `cmd.exe`.
- `dir`: specifies the working directory on the workers. Defaults to the host's current directory (as found by `pwd()`)
- `enable_threaded_blas`: if `true` then BLAS will run on multiple threads in added processes. Default is `false`.
- `exename`: name of the `julia` executable. Defaults to `"$(Sys.BINDIR)/julia"` or `"$(Sys.BINDIR)/julia-debug"` as the case may be. It is recommended that a common Julia version is used on all remote machines because serialization and code distribution might fail otherwise.

- exeflags: additional flags passed to the worker processes. It can either be a Cmd, a String holding one flag, or a collection of strings, with one element per flag. E.g. $--threads = autoproject =$., "--compile-trace=stderr" or ["--threads=auto", "--compile=all"].
- topology: Specifies how the workers connect to each other. Sending a message between unconnected workers results in an error.
  - topology=:all_to_all: All processes are connected to each other. The default.
  - topology=:master_worker: Only the driver process, i.e. pid 1 connects to the workers. The workers do not connect to each other.
  - topology=:custom: The launch method of the cluster manager specifies the connection topology via fields ident and connect_idents in WorkerConfig. A worker with a cluster manager identity ident will connect to all workers specified in connect_idents.
- lazy: Applicable only with topology=:all_to_all. If true, worker-worker connections are setup lazily, i.e. they are setup at the first instance of a remote call between workers. Default is true.
- env: provide an array of string pairs such as env=["JULIA_DEPOT_PATH"=>"/depot"] to request that environment variables are set on the remote machine. By default only the environment variable JULIA_WORKER_TIMEOUT is passed automatically from the local to the remote environment.
- cmdline_cookie: pass the authentication cookie via the --worker commandline option. The (more secure) default behaviour of passing the cookie via ssh stdio may hang with Windows workers that use older (pre-ConPTY) Julia or Windows versions, in which case cmdline_cookie=true offers a work-around.

Julia 1.6

The keyword arguments ssh, shell, env and cmdline_cookie were added in Julia 1.6.

Environment variables:

If the master process fails to establish a connection with a newly launched worker within 60.0 seconds, the worker treats it as a fatal situation and terminates. This timeout can be controlled via environment variable JULIA_WORKER_TIMEOUT. The value of JULIA_WORKER_TIMEOUT on the master process specifies the number of seconds a newly launched worker waits for connection establishment.

```
addprocs(np::Integer=Sys.CPU_THREADS; restrict=true, kwargs...) -> List of process identifiers
```

Launch np workers on the local host using the in-built LocalManager.

Local workers inherit the current package environment (i.e., active project, LOAD_PATH, and DEPOT_PATH) from the main process.

Warning

Note that workers do not run a ~/.julia/config/startup.jl startup script, nor do they synchronize their global state (such as command-line switches, global variables, new method definitions, and loaded modules) with any of the other running processes.

Keyword arguments:

- restrict::Bool: if true (default) binding is restricted to 127.0.0.1.

- `dir`, `exename`, `exeflags`, `env`, `topology`, `lazy`, `enable_threaded_blas`: same effect as for `SSHManager`, see documentation for `addprocs(machines::AbstractVector)`.

Julia 1.9

The inheriting of the package environment and the env keyword argument were added in Julia 1.9.

`Distributed.nprocs` – Function.

```
nprocs()
```

Get the number of available processes.

Examples

```
julia> nprocs()
3

julia> workers()
2-element Array{Int64,1}:
 2
 3
```

`Distributed.nworkers` – Function.

```
nworkers()
```

Get the number of available worker processes. This is one less than `nprocs()`. Equal to `nprocs()` if `nprocs() == 1`.

Examples

```
$ julia -p 2

julia> nprocs()
3

julia> nworkers()
2
```

`Distributed.procs` – Method.

```
procs()
```

Return a list of all process identifiers, including pid 1 (which is not included by `workers()`).

Examples

```
$ julia -p 2

julia> procs()
3-element Array{Int64,1}:
 1
 2
 3
```

`Distributed.procs` – Method.

```
procs(pid::Integer)
```

Return a list of all process identifiers on the same physical node. Specifically all workers bound to the same ip-address as `pid` are returned.

`Distributed.workers` – Function.

```
workers()
```

Return a list of all worker process identifiers.

Examples

```
$ julia -p 2

julia> workers()
2-element Array{Int64,1}:
 2
 3
```

`Distributed.rmprocs` – Function.

```
rmprocs(pids...; waitfor=typemax(Int))
```

Remove the specified workers. Note that only process 1 can add or remove workers.

Argument `waitfor` specifies how long to wait for the workers to shut down:

- If unspecified, `rmprocs` will wait until all requested `pids` are removed.
- An `ErrorException` is raised if all workers cannot be terminated before the requested `waitfor` seconds.
- With a `waitfor` value of 0, the call returns immediately with the workers scheduled for removal in a different task. The scheduled `Task` object is returned. The user should call `wait` on the task before invoking any other parallel calls.

Examples

```
$ julia -p 5

julia> t = rmprocs(2, 3, waitfor=0)
Task (runnable) @0x0000000107c718d0

julia> wait(t)

julia> workers()
3-element Array{Int64,1}:
 4
 5
 6
```

`Distributed.interrupt` – Function.

```
interrupt(pids::Integer...)
```

Interrupt the current executing task on the specified workers. This is equivalent to pressing Ctrl-C on the local machine. If no arguments are given, all workers are interrupted.

```
interrupt(pids::AbstractVector=workers())
```

Interrupt the current executing task on the specified workers. This is equivalent to pressing Ctrl-C on the local machine. If no arguments are given, all workers are interrupted.

`Distributed.myid` – Function.

```
myid()
```

Get the id of the current process.

Examples

```
julia> myid()
1

julia> remotecall_fetch(() -> myid(), 4)
4
```

`Distributed.pmap` – Function.

```
pmap(f, [::AbstractWorkerPool], c...; distributed=true, batch_size=1, on_error=nothing,
↪  retry_delays=[], retry_check=nothing) -> collection
```

Transform collection `c` by applying `f` to each element using available workers and tasks.

For multiple collection arguments, apply `f` elementwise.

Note that `f` must be made available to all worker processes; see Code Availability and Loading Packages for details.

If a worker pool is not specified all available workers will be used via a `CachingPool`.

By default, `pmap` distributes the computation over all specified workers. To use only the local process and distribute over tasks, specify `distributed=false`. This is equivalent to using `asyncmap`. For example, `pmap(f, c; distributed=false)` is equivalent to `asyncmap(f,c; ntasks=()->nworkers())`

`pmap` can also use a mix of processes and tasks via the `batch_size` argument. For batch sizes greater than 1, the collection is processed in multiple batches, each of length `batch_size` or less. A batch is sent as a single request to a free worker, where a local `asyncmap` processes elements from the batch using multiple concurrent tasks.

Any error stops `pmap` from processing the remainder of the collection. To override this behavior you can specify an error handling function via argument `on_error` which takes in a single argument, i.e., the exception. The function can stop the processing by rethrowing the error, or, to continue, return any value which is then returned inline with the results to the caller.

Consider the following two examples. The first one returns the exception object inline, the second a 0 in place of any exception:

```
julia> pmap(x->iseven(x) ? error("foo") : x, 1:4; on_error=identity)
4-element Array{Any,1}:
 1
  ErrorException("foo")
 3
  ErrorException("foo")

julia> pmap(x->iseven(x) ? error("foo") : x, 1:4; on_error=ex->0)
4-element Array{Int64,1}:
 1
 0
 3
 0
```

Errors can also be handled by retrying failed computations. Keyword arguments `retry_delays` and `retry_check` are passed through to `retry` as keyword arguments `delays` and `check` respectively. If batching is specified, and an entire batch fails, all items in the batch are retried.

Note that if both `on_error` and `retry_delays` are specified, the `on_error` hook is called before retrying. If `on_error` does not throw (or rethrow) an exception, the element will not be retried.

Example: On errors, retry `f` on an element a maximum of 3 times without any delay between retries.

```
pmap(f, c; retry_delays = zeros(3))
```

Example: Retry `f` only if the exception is not of type `InexactError`, with exponentially increasing delays up to 3 times. Return a `NaN` in place for all `InexactError` occurrences.

```
pmap(f, c; on_error = e->(isa(e, InexactError) ? NaN : rethrow()), retry_delays =
↪  ExponentialBackOff(n = 3))
```

`Distributed.RemoteException` – Type.

```
RemoteException(captured)
```

Exceptions on remote computations are captured and rethrown locally. A `RemoteException` wraps the `pid` of the worker and a captured exception. A `CapturedException` captures the remote exception and a serializable form of the call stack when the exception was raised.

`Distributed.ProcessExitedException` – Type.

```
ProcessExitedException(worker_id::Int)
```

After a client Julia process has exited, further attempts to reference the dead child will throw this exception.

`Distributed.Future` – Type.

```
Future(w::Int, rrid::RRID, v::Union{Some, Nothing}=nothing)
```

A `Future` is a placeholder for a single computation of unknown termination status and time. For multiple potential computations, see `RemoteChannel`. See `remoteref_id` for identifying an `AbstractRemoteRef`.

`Distributed.RemoteChannel` – Type.

```
RemoteChannel(pid::Integer=myid())
```

Make a reference to a `Channel{Any}(1)` on process `pid`. The default `pid` is the current process.

```
RemoteChannel(f::Function, pid::Integer=myid())
```

Create references to remote channels of a specific size and type. `f` is a function that when executed on `pid` must return an implementation of an `AbstractChannel`.

For example, `RemoteChannel(()->Channel{Int}(10), pid)`, will return a reference to a channel of type `Int` and size 10 on `pid`.

The default `pid` is the current process.

`Base.fetch` – Method.

```
fetch(x::Future)
```

Wait for and get the value of a `Future`. The fetched value is cached locally. Further calls to `fetch` on the same reference return the cached value. If the remote value is an exception, throws a `RemoteException` which captures the remote exception and backtrace.

`Base.fetch` – Method.

```
fetch(c::RemoteChannel)
```

Wait for and get a value from a `RemoteChannel`. Exceptions raised are the same as for a `Future`. Does not remove the item fetched.

```
fetch(x::Any)
```

Return `x`.

source

`Distributed.remotecall` – Method.

```
remotecall(f, id::Integer, args...; kwargs...) -> Future
```

Call a function `f` asynchronously on the given arguments on the specified process. Return a `Future`. Keyword arguments, if any, are passed through to `f`.

`Distributed.remotecall_wait` – Method.

```
remotecall_wait(f, id::Integer, args...; kwargs...)
```

Perform a faster `wait(remotecall(...))` in one message on the `Worker` specified by worker id `id`. Keyword arguments, if any, are passed through to `f`.

See also `wait` and `remotecall`.

`Distributed.remotecall_fetch` – Method.

```
remotecall_fetch(f, id::Integer, args...; kwargs...)
```

Perform `fetch(remotecall(...))` in one message. Keyword arguments, if any, are passed through to `f`. Any remote exceptions are captured in a `RemoteException` and thrown.

See also `fetch` and `remotecall`.

Examples

```
$ julia -p 2

julia> remotecall_fetch(sqrt, 2, 4)
2.0

julia> remotecall_fetch(sqrt, 2, -4)
ERROR: On worker 2:
DomainError with -4.0:
sqrt was called with a negative real argument but will only return a complex result if called
↪  with a complex argument. Try sqrt(Complex(x)).
...
```

`Distributed.remotecall_eval` – Function.

```
remotecall_eval(m::Module, procs, expression)
```

Execute an expression under module `m` on the processes specified in `procs`. Errors on any of the processes are collected into a `CompositeException` and thrown.

See also `@everywhere`.

`Distributed.remote_do` – Method.

```
remote_do(f, id::Integer, args...; kwargs...) -> nothing
```

Executes `f` on worker `id` asynchronously. Unlike `remotecall`, it does not store the result of computation, nor is there a way to wait for its completion.

A successful invocation indicates that the request has been accepted for execution on the remote node.

While consecutive `remotecalls` to the same worker are serialized in the order they are invoked, the order of executions on the remote worker is undetermined. For example, `remote_do(f1, 2); remotecall(f2, 2); remote_do(f3, 2)` will serialize the call to `f1`, followed by `f2` and `f3` in that order. However, it is not guaranteed that `f1` is executed before `f3` on worker 2.

Any exceptions thrown by `f` are printed to `stderr` on the remote worker.

Keyword arguments, if any, are passed through to `f`.

`Base.put!` – Method.

```
put!(rr::RemoteChannel, args...)
```

Store a set of values to the `RemoteChannel`. If the channel is full, blocks until space is available. Return the first argument.

`Base.put!` – Method.

```
put!(rr::Future, v)
```

Store a value to a Future rr. Futures are write-once remote references. A put! on an already set Future throws an Exception. All asynchronous remote calls return Futures and set the value to the return value of the call upon completion.

Base.take! – Method.

```
take!(rr::RemoteChannel, args...)
```

Fetch value(s) from a RemoteChannel rr, removing the value(s) in the process.

Base.isready – Method.

```
isready(rr::RemoteChannel, args...)
```

Determine whether a RemoteChannel has a value stored to it. Note that this function can cause race conditions, since by the time you receive its result it may no longer be true. However, it can be safely used on a Future since they are assigned only once.

Base.isready – Method.

```
isready(rr::Future)
```

Determine whether a Future has a value stored to it.

If the argument Future is owned by a different node, this call will block to wait for the answer. It is recommended to wait for rr in a separate task instead or to use a local Channel as a proxy:

```
p = 1
f = Future(p)
errormonitor(@async put!(f, remotecall_fetch(long_computation, p)))
isready(f)  # will not block
```

Distributed.AbstractWorkerPool – Type.

```
AbstractWorkerPool
```

Supertype for worker pools such as WorkerPool and CachingPool. An AbstractWorkerPool should implement:

- push! - add a new worker to the overall pool (available + busy)
- put! - put back a worker to the available pool
- take! - take a worker from the available pool (to be used for remote function execution)
- wait - block until a worker is available
- length - number of workers available in the overall pool
- isready - return false if a take! on the pool would block, else true

The default implementations of the above (on a AbstractWorkerPool) require fields

- channel::Channel{Int}
- workers::Set{Int}

where channel contains free worker pids and workers is the set of all workers associated with this pool.

`Distributed.WorkerPool` – Type.

```
WorkerPool(workers::Union{Vector{Int},AbstractRange{Int}})
```

Create a `WorkerPool` from a vector or range of worker ids.

Examples

```
$ julia -p 3

julia> WorkerPool([2, 3])
WorkerPool(Channel{Int64}(sz_max:9223372036854775807,sz_curr:2), Set([2, 3]),
↪  RemoteChannel{Channel{Any}}(1, 1, 6))

julia> WorkerPool(2:4)
WorkerPool(Channel{Int64}(sz_max:9223372036854775807,sz_curr:2), Set([4, 2, 3]),
↪  RemoteChannel{Channel{Any}}(1, 1, 7))
```

`Distributed.CachingPool` – Type.

```
CachingPool(workers::Vector{Int})
```

An implementation of an `AbstractWorkerPool`. `remote`, `remotecall_fetch`, `pmap` (and other remote calls which execute functions remotely) benefit from caching the serialized/deserialized functions on the worker nodes, especially closures (which may capture large amounts of data).

The remote cache is maintained for the lifetime of the returned `CachingPool` object. To clear the cache earlier, use `clear!(pool)`.

For global variables, only the bindings are captured in a closure, not the data. `let` blocks can be used to capture global data.

Examples

```
const foo = rand(10^8);
wp = CachingPool(workers())
let foo = foo
    pmap(i -> sum(foo) + i, wp, 1:100);
end
```

The above would transfer `foo` only once to each worker.

`Distributed.default_worker_pool` – Function.

```
default_worker_pool()
```

`AbstractWorkerPool` containing idle `workers` - used by `remote(f)` and `pmap` (by default). Unless one is explicitly set via `default_worker_pool!(pool)`, the default worker pool is initialized to a `WorkerPool`.

Examples

```
$ julia -p 3

julia> default_worker_pool()
WorkerPool(Channel{Int64}(sz_max:9223372036854775807,sz_curr:3), Set([4, 2, 3]),
↪  RemoteChannel{Channel{Any}}(1, 1, 4))
```

`Distributed.clear!` – Function.

```
clear!(syms, pids=workers(); mod=Main)
```

Clears global bindings in modules by initializing them to `nothing`. `syms` should be of type `Symbol` or a collection of `Symbols` . `pids` and `mod` identify the processes and the module in which global variables are to be reinitialized. Only those names found to be defined under `mod` are cleared.

An exception is raised if a global constant is requested to be cleared.

```
clear!(pool::CachingPool) -> pool
```

Removes all cached functions from all participating workers.

`Distributed.remote` – Function.

```
remote([p::AbstractWorkerPool], f) -> Function
```

Return an anonymous function that executes function `f` on an available worker (drawn from `WorkerPool` `p` if provided) using `remotecall_fetch`.

`Distributed.remotecall` – Method.

```
remotecall(f, pool::AbstractWorkerPool, args...; kwargs...) -> Future
```

`WorkerPool` variant of `remotecall(f, pid, ....)`. Wait for and take a free worker from `pool` and perform a `remotecall` on it.

Examples

```
$ julia -p 3

julia> wp = WorkerPool([2, 3]);

julia> A = rand(3000);

julia> f = remotecall(maximum, wp, A)
Future(2, 1, 6, nothing)
```

In this example, the task ran on pid 2, called from pid 1.

`Distributed.remotecall_wait` – Method.

```
remotecall_wait(f, pool::AbstractWorkerPool, args...; kwargs...) -> Future
```

`WorkerPool` variant of `remotecall_wait(f, pid, ....)`. Wait for and take a free worker from `pool` and perform a `remotecall_wait` on it.

Examples

```
$ julia -p 3

julia> wp = WorkerPool([2, 3]);

julia> A = rand(3000);

julia> f = remotecall_wait(maximum, wp, A)
Future(3, 1, 9, nothing)

julia> fetch(f)
0.9995177101692958
```

`Distributed.remotecall_fetch` – Method.

```
remotecall_fetch(f, pool::AbstractWorkerPool, args...; kwargs...) -> result
```

`WorkerPool` variant of `remotecall_fetch(f, pid, ....)`. Waits for and takes a free worker from `pool` and performs a `remotecall_fetch` on it.

Examples

```
$ julia -p 3

julia> wp = WorkerPool([2, 3]);

julia> A = rand(3000);

julia> remotecall_fetch(maximum, wp, A)
0.9995177101692958
```

`Distributed.remote_do` – Method.

```
remote_do(f, pool::AbstractWorkerPool, args...; kwargs...) -> nothing
```

`WorkerPool` variant of `remote_do(f, pid, ....)`. Wait for and take a free worker from `pool` and perform a `remote_do` on it.

Note that it's not possible to wait for the result of a `remote_do()` to finish so the worker will immediately be put back in the pool (i.e. potentially causing oversubscription).

`Distributed.@spawn` – Macro.

```
@spawn expr
```

Create a closure around an expression and run it on an automatically-chosen process, returning a `Future` to the result. This macro is deprecated; `@spawnat :any expr` should be used instead.

Examples

```
julia> addprocs(3);

julia> f = @spawn myid()
Future(2, 1, 5, nothing)
```

```
julia> fetch(f)
2

julia> f = @spawn myid()
Future(3, 1, 7, nothing)

julia> fetch(f)
3
```

Julia 1.3

As of Julia 1.3 this macro is deprecated. Use `@spawnat :any` instead.

`Distributed.@spawnat` – Macro.

```
@spawnat p expr
```

Create a closure around an expression and run the closure asynchronously on process p. Return a `Future` to the result.

If p is the quoted literal symbol `:any`, then the system will pick a processor to use automatically. Using `:any` will not apply any form of load-balancing, consider using a `WorkerPool` and `remotecall(f, ::WorkerPool)` if you need load-balancing.

Examples

```
julia> addprocs(3);

julia> f = @spawnat 2 myid()
Future(2, 1, 3, nothing)

julia> fetch(f)
2

julia> f = @spawnat :any myid()
Future(3, 1, 7, nothing)

julia> fetch(f)
3
```

Julia 1.3

The `:any` argument is available as of Julia 1.3.

`Distributed.@fetch` – Macro.

```
@fetch expr
```

Equivalent to `fetch(@spawnat :any expr)`. See `fetch` and `@spawnat`.

Examples

```
julia> addprocs(3);

julia> @fetch myid()
2

julia> @fetch myid()
3

julia> @fetch myid()
4

julia> @fetch myid()
2
```

`Distributed.@fetchfrom` – Macro.

```
@fetchfrom
```

Equivalent to `fetch(@spawnat p expr)`. See `fetch` and `@spawnat`.

Examples

```
julia> addprocs(3);

julia> @fetchfrom 2 myid()
2

julia> @fetchfrom 4 myid()
4
```

`Distributed.@distributed` – Macro.

```
@distributed
```

A distributed memory, parallel for loop of the form :

```
@distributed [reducer] for var = range
    body
end
```

The specified range is partitioned and locally executed across all workers. In case an optional reducer function is specified, `@distributed` performs local reductions on each worker with a final reduction on the calling process.

Note that without a reducer function, `@distributed` executes asynchronously, i.e. it spawns independent tasks on all available workers and returns immediately without waiting for completion. To wait for completion, prefix the call with `@sync`, like :

```
@sync @distributed for var = range
    body
end
```

`Distributed.@everywhere` – Macro.

```
@everywhere [procs()] expr
```

Execute an expression under `Main` on all `procs`. Errors on any of the processes are collected into a `CompositeException` and thrown. For example:

```
@everywhere bar = 1
```

will define `Main.bar` on all current processes. Any processes added later (say with `addprocs()`) will not have the expression defined.

Unlike `@spawnat`, `@everywhere` does not capture any local variables. Instead, local variables can be broadcast using interpolation:

```
foo = 1
@everywhere bar = $foo
```

The optional argument `procs` allows specifying a subset of all processes to have execute the expression.

Similar to calling `remotecall_eval(Main, procs, expr)`, but with two extra features:

- `using` and `import` statements run on the calling process first, to ensure packages are precompiled.
- The current source file path used by `include` is propagated to other processes.

`Distributed.remoteref_id` – Function.

```
remoteref_id(r::AbstractRemoteRef) -> RRID
```

`Futures` and `RemoteChannels` are identified by fields:

- `where` - refers to the node where the underlying object/storage referred to by the reference actually exists.
- `whence` - refers to the node the remote reference was created from. Note that this is different from the node where the underlying object referred to actually exists. For example calling `RemoteChannel(2)` from the master process would result in a `where` value of 2 and a `whence` value of 1.
- `id` is unique across all references created from the worker specified by `whence`.

Taken together, `whence` and `id` uniquely identify a reference across all workers.

`remoteref_id` is a low-level API which returns a `RRID` object that wraps `whence` and `id` values of a remote reference.

`Distributed.channel_from_id` – Function.

```
channel_from_id(id) -> c
```

A low-level API which returns the backing `AbstractChannel` for an `id` returned by `remoteref_id`. The call is valid only on the node where the backing channel exists.

`Distributed.worker_id_from_socket` – Function.

```
worker_id_from_socket(s) -> pid
```

A low-level API which, given a `IO` connection or a `Worker`, returns the `pid` of the worker it is connected to. This is useful when writing custom `serialize` methods for a type, which optimizes the data written out depending on the receiving process id.

`Distributed.cluster_cookie` – Method.

```
cluster_cookie() -> cookie
```

Return the cluster cookie.

`Distributed.cluster_cookie` – Method.

```
cluster_cookie(cookie) -> cookie
```

Set the passed cookie as the cluster cookie, then returns it.

71.1 Cluster Manager Interface

This interface provides a mechanism to launch and manage Julia workers on different cluster environments. There are two types of managers present in Base: `LocalManager`, for launching additional workers on the same host, and `SSHManager`, for launching on remote hosts via `ssh`. TCP/IP sockets are used to connect and transport messages between processes. It is possible for Cluster Managers to provide a different transport.

`Distributed.ClusterManager` – Type.

```
ClusterManager
```

Supertype for cluster managers, which control workers processes as a cluster. Cluster managers implement how workers can be added, removed and communicated with. `SSHManager` and `LocalManager` are subtypes of this.

`Distributed.WorkerConfig` – Type.

```
WorkerConfig
```

Type used by `ClusterManager`s to control workers added to their clusters. Some fields are used by all cluster managers to access a host:

- `io` – the connection used to access the worker (a subtype of `IO` or `Nothing`)
- `host` – the host address (either a `String` or `Nothing`)
- `port` – the port on the host used to connect to the worker (either an `Int` or `Nothing`)

Some are used by the cluster manager to add workers to an already-initialized host:

- `count` – the number of workers to be launched on the host
- `exename` – the path to the Julia executable on the host, defaults to `"$(Sys.BINDIR)/julia"` or `"$(Sys.BINDIR)/julia-debug"`
- `exeflags` – flags to use when launching Julia remotely

The `userdata` field is used to store information for each worker by external managers.

Some fields are used by `SSHManager` and similar managers:

- `tunnel` – `true` (use tunneling), `false` (do not use tunneling), or `nothing` (use default for the manager)
- `multiplex` – `true` (use SSH multiplexing for tunneling) or `false`
- `forward` – the forwarding option used for `-L` option of ssh
- `bind_addr` – the address on the remote host to bind to
- `sshflags` – flags to use in establishing the SSH connection
- `max_parallel` – the maximum number of workers to connect to in parallel on the host

Some fields are used by both `LocalManagers` and `SSHManagers`:

- `connect_at` – determines whether this is a worker-to-worker or driver-to-worker setup call
- `process` – the process which will be connected (usually the manager will assign this during `addprocs`)
- `ospid` – the process ID according to the host OS, used to interrupt worker processes
- `environ` – private dictionary used to store temporary information by Local/SSH managers
- `ident` – worker as identified by the `ClusterManager`
- `connect_idents` – list of worker ids the worker must connect to if using a custom topology
- `enable_threaded_blas` – `true`, `false`, or `nothing`, whether to use threaded BLAS or not on the workers

`Distributed.launch` – Function.

```
launch(manager::ClusterManager, params::Dict, launched::Array, launch_ntfy::Condition)
```

Implemented by cluster managers. For every Julia worker launched by this function, it should append a `WorkerConfig` entry to `launched` and notify `launch_ntfy`. The function MUST exit once all workers, requested by `manager` have been launched. `params` is a dictionary of all keyword arguments `addprocs` was called with.

`Distributed.manage` – Function.

```
manage(manager::ClusterManager, id::Integer, config::WorkerConfig. op::Symbol)
```

Implemented by cluster managers. It is called on the master process, during a worker's lifetime, with appropriate op values:

- with `:register`/`:deregister` when a worker is added / removed from the Julia worker pool.
- with `:interrupt` when `interrupt(workers)` is called. The `ClusterManager` should signal the appropriate worker with an interrupt signal.
- with `:finalize` for cleanup purposes.

`Base.kill` – Method.

```
kill(manager::ClusterManager, pid::Int, config::WorkerConfig)
```

Implemented by cluster managers. It is called on the master process, by rmprocs. It should cause the remote worker specified by pid to exit. kill(manager::ClusterManager.....) executes a remote exit() on pid.

Sockets.connect – Method.

```
connect(manager::ClusterManager, pid::Int, config::WorkerConfig) -> (instrm::IO, outstrm::IO)
```

Implemented by cluster managers using custom transports. It should establish a logical connection to worker with id pid, specified by config and return a pair of IO objects. Messages from pid to current process will be read off instrm, while messages to be sent to pid will be written to outstrm. The custom transport implementation must ensure that messages are delivered and received completely and in order. connect(manager::ClusterManager.....) sets up TCP/IP socket connections in-between workers.

Distributed.init_worker – Function.

```
init_worker(cookie::AbstractString, manager::ClusterManager=DefaultClusterManager())
```

Called by cluster managers implementing custom transports. It initializes a newly launched process as a worker. Command line argument --worker[=<cookie>] has the effect of initializing a process as a worker using TCP/IP sockets for transport. cookie is a cluster_cookie.

Distributed.start_worker – Function.

```
start_worker([out::IO=stdout], cookie::AbstractString=readline(stdin); close_stdin::Bool=true,
↪  stderr_to_stdout::Bool=true)
```

start_worker is an internal function which is the default entry point for worker processes connecting via TCP/IP. It sets up the process as a Julia cluster worker.

host:port information is written to stream out (defaults to stdout).

The function reads the cookie from stdin if required, and listens on a free port (or if specified, the port in the --bind-to command line option) and schedules tasks to process incoming TCP connections and requests. It also (optionally) closes stdin and redirects stderr to stdout.

It does not return.

Distributed.process_messages – Function.

```
process_messages(r_stream::IO, w_stream::IO, incoming::Bool=true)
```

Called by cluster managers using custom transports. It should be called when the custom transport implementation receives the first message from a remote worker. The custom transport must manage a logical connection to the remote worker and provide two IO objects, one for incoming messages and the other for messages addressed to the remote worker. If incoming is true, the remote peer initiated the connection. Whichever of the pair initiates the connection sends the cluster cookie and its Julia version number to perform the authentication handshake.

See also cluster_cookie.

Distributed.default_addprocs_params – Function.

```
default_addprocs_params(mgr::ClusterManager) -> Dict{Symbol, Any}
```

Implemented by cluster managers. The default keyword parameters passed when calling `addprocs(mgr)`. The minimal set of options is available by calling `default_addprocs_params()`

Chapter 72

Downloads

`Downloads.download` – Function.

```
download(url, [ output = tempname() ];
    [ method = "GET", ]
    [ headers = <none>, ]
    [ timeout = <none>, ]
    [ progress = <none>, ]
    [ verbose = false, ]
    [ debug = <none>, ]
    [ downloader = <default>, ]
) -> output

    url        :: AbstractString
    output     :: Union{AbstractString, AbstractCmd, IO}
    method     :: AbstractString
    headers    :: Union{AbstractVector, AbstractDict}
    timeout    :: Real
    progress   :: (total::Integer, now::Integer) --> Any
    verbose    :: Bool
    debug      :: (type, message) --> Any
    downloader :: Downloader
```

Download a file from the given url, saving it to `output` or if not specified, a temporary path. The `output` can also be an `IO` handle, in which case the body of the response is streamed to that handle and the handle is returned. If `output` is a command, the command is run and output is sent to it on stdin.

If the `downloader` keyword argument is provided, it must be a `Downloader` object. Resources and connections will be shared between downloads performed by the same `Downloader` and cleaned up automatically when the object is garbage collected or there have been no downloads performed with it for a grace period. See `Downloader` for more info about configuration and usage.

If the `headers` keyword argument is provided, it must be a vector or dictionary whose elements are all pairs of strings. These pairs are passed as headers when downloading URLs with protocols that supports them, such as HTTP/S.

The `timeout` keyword argument specifies a timeout for the download to complete in seconds, with a resolution of milliseconds. By default no timeout is set, but this can also be explicitly requested by passing a timeout value of `Inf`. Separately, if 20 seconds elapse without receiving any data, the download will timeout. See extended help for how to disable this timeout.

If the `progress` keyword argument is provided, it must be a callback function which will be called whenever there are updates about the size and status of the ongoing download. The callback must take two integer arguments: `total` and `now` which are the total size of the download in bytes, and the number of bytes which have been downloaded so far. Note that `total` starts out as zero and remains zero until the server gives an indication of the total size of the download (e.g. with a `Content-Length` header), which may never happen. So a well-behaved progress callback should handle a total size of zero gracefully.

If the `verbose` option is set to true, `libcurl`, which is used to implement the download functionality will print debugging information to `stderr`. If the `debug` option is set to a function accepting two `String` arguments, then the verbose option is ignored and instead the data that would have been printed to `stderr` is passed to the debug callback with `type` and `message` arguments. The `type` argument indicates what kind of event has occurred, and is one of: `TEXT`, `HEADER IN`, `HEADER OUT`, `DATA IN`, `DATA OUT`, `SSL DATA IN` or `SSL DATA OUT`. The `message` argument is the description of the debug event.

Extended Help

For further customization, use a `Downloader` and `easy_hooks`. For example, to disable the 20 second timeout when no data is received, you may use the following:

```
downloader = Downloads.Downloader()
downloader.easy_hook = (easy, info) -> Downloads.Curl.setopt(easy,
↪  Downloads.Curl.CURLOPT_LOW_SPEED_TIME, 0)

Downloads.download("https://httpbingo.julialang.org/delay/30"; downloader)
```

`Downloads.request` – Function.

```
request(url;
    [ input = <none>, ]
    [ output = <none>, ]
    [ method = input ? "PUT" : output ? "GET" : "HEAD", ]
    [ headers = <none>, ]
    [ timeout = <none>, ]
    [ progress = <none>, ]
    [ verbose = false, ]
    [ debug = <none>, ]
    [ throw = true, ]
    [ downloader = <default>, ]
    [ interrupt = <none>, ]
) -> Union{Response, RequestError}

    url        :: AbstractString
    input      :: Union{AbstractString, AbstractCmd, IO}
    output     :: Union{AbstractString, AbstractCmd, IO}
    method     :: AbstractString
    headers    :: Union{AbstractVector, AbstractDict}
    timeout    :: Real
    progress   :: (dl_total, dl_now, ul_total, ul_now) --> Any
    verbose    :: Bool
    debug      :: (type, message) --> Any
    throw      :: Bool
    downloader :: Downloader
    interrupt  :: Base.Event
```

Make a request to the given url, returning a `Response` object capturing the status, headers and other information about the response. The body of the response is written to `output` if specified and discarded otherwise. For HTTP/S requests, if an `input` stream is given, a `PUT` request is made; otherwise if an `output` stream is given, a `GET` request is made; if neither is given a `HEAD` request is made. For other protocols, appropriate default methods are used based on what combination of input and output are requested. The following options differ from the `download` function:

- `input` allows providing a request body; if provided default to `PUT` request
- `progress` is a callback taking four integers for upload and download progress
- `throw` controls whether to throw or return a `RequestError` on request error

Note that unlike `download` which throws an error if the requested URL could not be downloaded (indicated by non-2xx status code), `request` returns a `Response` object no matter what the status code of the response is. If there is an error with getting a response at all, then a `RequestError` is thrown or returned.

If the `interrupt` keyword argument is provided, it must be a `Base.Event` object. If the event is triggered while the request is in progress, the request will be cancelled and an error will be thrown. This can be used to interrupt a long running request, for example if the user wants to cancel a download.

`Downloads.Response` – Type.

```
struct Response
    proto   :: String
    url     :: String
    status  :: Int
    message :: String
    headers :: Vector{Pair{String,String}}
end
```

`Response` is a type capturing the properties of a successful response to a request as an object. It has the following fields:

- `proto`: the protocol that was used to get the response
- `url`: the URL that was ultimately requested after following redirects
- `status`: the status code of the response, indicating success, failure, etc.
- `message`: a textual message describing the nature of the response
- `headers`: any headers that were returned with the response

The meaning and availability of some of these responses depends on the protocol used for the request. For many protocols, including HTTP/S and S/FTP, a 2xx status code indicates a successful response. For responses in protocols that do not support headers, the headers vector will be empty. HTTP/2 does not include a status message, only a status code, so the message will be empty.

`Downloads.RequestError` – Type.

```
struct RequestError <: ErrorException
    url      :: String
    code     :: Int
    message  :: String
    response :: Response
end
```

`RequestError` is a type capturing the properties of a failed response to a request as an exception object:

- `url`: the original URL that was requested without any redirects
- `code`: the libcurl error code; `0` if a protocol-only error occurred
- `message`: the libcurl error message indicating what went wrong
- `response`: response object capturing what response info is available

The same `RequestError` type is thrown by `download` if the request was successful but there was a protocol-level error indicated by a status code that is not in the 2xx range, in which case `code` will be zero and the `message` field will be the empty string. The `request` API only throws a `RequestError` if the libcurl error `code` is non-zero, in which case the included `response` object is likely to have a `status` of zero and an empty message. There are, however, situations where a curl-level error is thrown due to a protocol error, in which case both the inner and outer code and message may be of interest.

`Downloads.Downloader` – Type.

```
Downloader(; [ grace::Real = 30 ])
```

`Downloader` objects are used to perform individual `download` operations. Connections, name lookups and other resources are shared within a `Downloader`. These connections and resources are cleaned up after a configurable grace period (default: 30 seconds) since anything was downloaded with it, or when it is garbage collected, whichever comes first. If the grace period is set to zero, all resources will be cleaned up immediately as soon as there are no more ongoing downloads in progress. If the grace period is set to `Inf` then resources are not cleaned up until `Downloader` is garbage collected.

`Libdl` – Module.

The Libdl module in Julia provides specialized and lower-level facilities for dynamic linking with shared libraries. While Julia inherently supports linking to runtime shared libraries through the `ccall` intrinsic, `Libdl` extends this capability by offering additional, more granular control. It enables users to search for shared libraries both in memory and the filesystem, manually load them with specific runtime linker options, and look up library symbols as low-level pointers.

Chapter 73

Dynamic Linker

Base.Libc.Libdl.dlopen – Function.

```
dlopen(libfile::AbstractString [, flags::Integer]; throw_error:Bool = true)
```

Load a shared library, returning an opaque handle.

The extension given by the constant `dlext` (`.so`, `.dll`, or `.dylib`) can be omitted from the `libfile` string, as it is automatically appended if needed. If `libfile` is not an absolute path name, then the paths in the array `DL_LOAD_PATH` are searched for `libfile`, followed by the system load path.

The optional flags argument is a bitwise-or of zero or more of `RTLD_LOCAL`, `RTLD_GLOBAL`, `RTLD_LAZY`, `RTLD_NOW`, `RTLD_NODELETE`, `RTLD_NOLOAD`, `RTLD_DEEPBIND`, and `RTLD_FIRST`. These are converted to the corresponding flags of the POSIX (and/or GNU libc and/or MacOS) dlopen command, if possible, or are ignored if the specified functionality is not available on the current platform. The default flags are platform specific. On MacOS the default `dlopen` flags are `RTLD_LAZY|RTLD_DEEPBIND|RTLD_GLOBAL` while on other platforms the defaults are `RTLD_LAZY|RTLD_DEEPBIND|RTLD_LOCAL`. An important usage of these flags is to specify non default behavior for when the dynamic library loader binds library references to exported symbols and if the bound references are put into process local or global scope. For instance `RTLD_LAZY|RTLD_DEEPBIND|RTLD_GLOBAL` allows the library's symbols to be available for usage in other shared libraries, addressing situations where there are dependencies between shared libraries.

If the library cannot be found, this method throws an error, unless the keyword argument `throw_error` is set to `false`, in which case this method returns `nothing`.

Note

From Julia 1.6 on, this method replaces paths starting with `@executable_path/` with the path to the Julia executable, allowing for relocatable relative-path loads. In Julia 1.5 and earlier, this only worked on macOS.

source

Base.Libc.Libdl.dlopen_e – Function.

```
dlopen_e(libfile::AbstractString [, flags::Integer])
```

Similar to `dlopen`, except returns `C_NULL` instead of raising errors. This method is now deprecated in favor of `dlopen(libfile::AbstractString [, flags::Integer]; throw_error=false)`.

source

Base.Libc.Libdl.RTLD_NOW – Constant.

```
RTLD_DEEPBIND
RTLD_FIRST
RTLD_GLOBAL
RTLD_LAZY
RTLD_LOCAL
RTLD_NODELETE
RTLD_NOLOAD
RTLD_NOW
```

Enum constant for dlopen. See your platform man page for details, if applicable.

source

Base.Libc.Libdl.dlsym – Function.

```
dlsym(handle, sym; throw_error::Bool = true)
```

Look up a symbol from a shared library handle, return callable function pointer on success.

If the symbol cannot be found, this method throws an error, unless the keyword argument `throw_error` is set to `false`, in which case this method returns `nothing`.

source

Base.Libc.Libdl.dlsym_e – Function.

```
dlsym_e(handle, sym)
```

Look up a symbol from a shared library handle, silently return `C_NULL` on lookup failure. This method is now deprecated in favor of `dlsym(handle, sym; throw_error=false)`.

source

Base.Libc.Libdl.dlclose – Function.

```
dlclose(::Nothing)
```

For the very common pattern usage pattern of

```
try
    hdl = dlopen(library_name)
    ... do something
finally
    dlclose(hdl)
end
```

We define a `dlclose()` method that accepts a parameter of type `Nothing`, so that user code does not have to change its behavior for the case that `library_name` was not found.

source

```
dlclose(handle)
```

Close shared library referenced by handle.

source

`Base.Libc.Libdl.dlext` – Constant.

```
dlext
```

File extension for dynamic libraries (e.g. dll, dylib, so) on the current platform.

source

`Base.Libc.Libdl.dllist` – Function.

```
dllist()
```

Return the paths of dynamic libraries currently loaded in a `Vector{String}`.

source

`Base.Libc.Libdl.dlpath` – Function.

```
dlpath(libname::Union{AbstractString, Symbol})
```

Get the full path of the library `libname`.

Examples

```
julia> dlpath("libjulia")
```

source

```
dlpath(handle::Ptr{Cvoid})
```

Given a library `handle` from `dlopen`, return the full path.

source

`Base.Libc.Libdl.find_library` – Function.

```
find_library(names [, locations])
```

Searches for the first library in `names` in the paths in the `locations` list, `DL_LOAD_PATH`, or system library paths (in that order) which can successfully be dlopen'd. On success, the return value will be one of the names (potentially prefixed by one of the paths in locations). This string can be assigned to a `global const` and used as the library name in future `ccall`'s. On failure, it returns the empty string.

source

`Base.DL_LOAD_PATH` – Constant.

```
DL_LOAD_PATH
```

When calling `dlopen`, the paths in this list will be searched first, in order, before searching the system locations for a valid library handle.

source

Chapter 74

Lazy Library Loading

`Base.Libc.Libdl.LazyLibrary` – Type.

```
LazyLibrary(name; flags = <default dlopen flags>,
            dependencies = LazyLibrary[], on_load_callback = nothing)
```

Represents a lazily-loaded shared library that delays loading itself and its dependencies until first use in a `ccall()`, `@ccall`, `dlopen()`, `dlsym()`, `dlpath()`, or `cglobal()`. This is a thread-safe mechanism for on-demand library initialization.

Arguments

- `name`: Library name (or lazy path computation) as a `String`, `LazyLibraryPath`, or `BundledLazyLibraryPath`.
- `flags`: Optional dlopen flags (default: `RTLD_LAZY | RTLD_DEEPBIND`). See `dlopen`.
- `dependencies`: Vector of `LazyLibrary` object references to load before this one.
- `on_load_callback`: Optional function to run arbitrary code on first load (use sparingly, as it is not expected that `ccall()` should result in large amounts of Julia code being run. You may call `ccall()` from within the `on_load_callback` but only for the current library and its dependencies, and user should not call `wait()` on any tasks within the on load callback as they may deadlock).

The dlopen operation is thread-safe: only one thread loads the library, acquired after the release store of the reference to each dependency from loading of each dependency. Other tasks block until loading completes. The handle is then cached and reused for all subsequent calls (there is no dlclose for lazy library and dlclose should not be called on the returned handled).

Examples

```
# Basic usage
const mylib = LazyLibrary("libmylib")
@ccall mylib.myfunc(42::Cint)::Cint

# With dependencies
const libfoo = LazyLibrary("libfoo")
const libbar = LazyLibrary("libbar"; dependencies=[libfoo])
```

For more examples including platform-specific libraries, lazy path construction, and migration from `__init__()` patterns, see the manual section on Using LazyLibrary for Lazy Loading.

Julia 1.11

LazyLibrary was added in Julia 1.11.

See also LazyLibraryPath, BundledLazyLibraryPath, dlopen, dlsym, add_dependency!.

source

Base.Libc.Libdl.LazyLibraryPath – Type.

```
LazyLibraryPath(path_pieces...)
```

Helper type for lazily constructed library paths for use with LazyLibrary. Path pieces are stored unevaluated and joined with joinpath() when the library is first accessed. Arguments must be able to have string() called on them.

Example

```
const mylib = LazyLibrary(LazyLibraryPath(artifact_dir, "lib", "libmylib.so.1.2.3"))
```

Julia 1.11

LazyLibraryPath was added in Julia 1.11.

See also LazyLibrary, BundledLazyLibraryPath.

source

Base.Libc.Libdl.BundledLazyLibraryPath – Function.

```
BundledLazyLibraryPath(subpath)
```

Helper type for lazily constructed library paths within the Julia distribution. Constructs paths relative to Julia's private shared library directory.

Primarily used by Julia's standard library. For example:

```
const libgmp = LazyLibrary(BundledLazyLibraryPath("libgmp.so.10"))
```

Julia 1.11

BundledLazyLibraryPath was added in Julia 1.11.

See also LazyLibrary, LazyLibraryPath.

source

Base.Libc.Libdl.add_dependency! – Function.

```
add_dependency!(library::LazyLibrary, dependency::LazyLibrary)
```

Dynamically add a dependency that must be loaded before library. Only needed when dependencies cannot be determined at construction time.

Warning

Dependencies added with this function are **ephemeral** and only persist within the current process. They will not persist across precompilation boundaries.

Prefer specifying dependencies in the `LazyLibrary` constructor when possible.

Julia 1.11

`add_dependency!` was added in Julia 1.11.

See also `LazyLibrary`.

source

Chapter 75

File Events

`FileWatching.poll_fd` – Function.

```
poll_fd(fd, timeout_s::Real=-1; readable=false, writable=false)
```

Monitor a file descriptor `fd` for changes in the read or write availability, and with a timeout given by `timeout_s` seconds.

The keyword arguments determine which of read and/or write status should be monitored; at least one of them must be set to `true`.

The returned value is an object with boolean fields `readable`, `writable`, and `timedout`, giving the result of the polling.

This is a thin wrapper over calling `wait` on a `FDWatcher`, which implements the functionality but requires the user to call `close` manually when finished with it, or risk serious crashes.

`FileWatching.poll_file` – Function.

```
poll_file(path::AbstractString, interval_s::Real=5.007, timeout_s::Real=-1) ->
↪  (previous::StatStruct, current)
```

Monitor a file for changes by polling every `interval_s` seconds until a change occurs or `timeout_s` seconds have elapsed. The `interval_s` should be a long period; the default is 5.007 seconds.

Returns a pair of status objects `(previous, current)` when a change is detected. The `previous` status is always a `StatStruct`, but it may have all of the fields zeroed (indicating the file didn't previously exist, or wasn't previously accessible).

The `current` status object may be a `StatStruct`, an `EOFError` (indicating the timeout elapsed), or some other `Exception` subtype (if the `stat` operation failed: for example, if the path does not exist).

To determine when a file was modified, compare `!(current isa StatStruct && prev == current)` to detect notification of changes to the mtime or inode. However, using `watch_file` for this operation is preferred, since it is more reliable and efficient, although in some situations it may not be available.

This is a thin wrapper over calling `wait` on a `PollingFileWatcher`, which implements the functionality, but this function has a small race window between consecutive calls to `poll_file` where the file might change without being detected.

`FileWatching.watch_file` – Function.

```
watch_file(path::AbstractString, timeout_s::Real=-1)
```

Watch file or directory `path` for changes until a change occurs or `timeout_s` seconds have elapsed. This function does not poll the file system and instead uses platform-specific functionality to receive notifications from the operating system (e.g. via inotify on Linux). See the NodeJS documentation linked below for details.

The returned value is an object with boolean fields `renamed`, `changed`, and `timedout`, giving the result of watching the file.

This behavior of this function varies slightly across platforms. See https://nodejs.org/api/fs.html#fs_caveats for more detailed information.

This is a thin wrapper over calling `wait` on a `FileMonitor`. This function has a small race window between consecutive calls to `watch_file` where the file might change without being detected. To avoid this race, use

```
fm = FileMonitor(path)
wait(fm)
```

directly, re-using the same `fm` each time you `wait`.

`FileWatching.watch_folder` – Function.

```
watch_folder(path::AbstractString, timeout_s::Real=-1)
```

Watch a file or directory `path` for changes until a change has occurred or `timeout_s` seconds have elapsed. This function does not poll the file system and instead uses platform-specific functionality to receive notifications from the operating system (e.g. via inotify on Linux). See the NodeJS documentation linked below for details.

This will continuing tracking changes for `path` in the background until `unwatch_folder` is called on the same `path`.

The returned value is an pair where the first field is the name of the changed file (if available) and the second field is an object with boolean fields `renamed`, `changed`, and `timedout`, giving the event.

This behavior of this function varies slightly across platforms. See https://nodejs.org/api/fs.html#fs_caveats for more detailed information.

This function is a thin wrapper over calling `wait` on a `FolderMonitor`, with added timeout support.

`FileWatching.unwatch_folder` – Function.

```
unwatch_folder(path::AbstractString)
```

Stop background tracking of changes for `path`. It is not recommended to do this while another task is waiting for `watch_folder` to return on the same path, as the result may be unpredictable.

`FileWatching.FileMonitor` – Type.

```
FileMonitor(path::AbstractString)
```

Watch file or directory `path` (which must exist) for changes until a change occurs. This function does not poll the file system and instead uses platform-specific functionality to receive notifications from the operating system (e.g. via inotify on Linux). See the NodeJS documentation linked below for details.

`fm = FileMonitor(path)` acts like an auto-reset Event, so `wait(fm)` blocks until there has been at least one event in the file originally at the given path and then returns an object with boolean fields `renamed`, `changed`, `timedout` summarizing all changes that have occurred since the last call to `wait` returned.

This behavior of this function varies slightly across platforms. See https://nodejs.org/api/fs.html#fs_caveats for more detailed information.

`FileWatching.FolderMonitor` – Type.

```
FolderMonitor(folder::AbstractString)
```

Watch a file or directory `path` for changes until a change has occurred. This function does not poll the file system and instead uses platform-specific functionality to receive notifications from the operating system (e.g. via inotify on Linux). See the NodeJS documentation linked below for details.

This acts similar to a Channel, so calling `take!` (or `wait`) blocks until some change has occurred. The `wait` function will return a pair where the first field is the name of the changed file (if available) and the second field is an object with boolean fields `renamed` and `changed`, giving the event that occurred on it.

This behavior of this function varies slightly across platforms. See https://nodejs.org/api/fs.html#fs_caveats for more detailed information.

`FileWatching.PollingFileWatcher` – Type.

```
PollingFileWatcher(path::AbstractString, interval_s::Real=5.007)
```

Monitor a file for changes by polling `stat` every `interval_s` seconds until a change occurs or `timeout_s` seconds have elapsed. The `interval_s` should be a long period; the default is 5.007 seconds. Call `stat` on it to get the most recent, but old, result.

This acts like an auto-reset Event, so calling `wait` blocks until the `stat` result has changed since the previous value captured upon entry to the `wait` call. The `wait` function will return a pair of status objects `(previous, current)` once any `stat` change is detected since the previous time that `wait` was called. The `previous` status is always a `StatStruct`, but it may have all of the fields zeroed (indicating the file didn't previously exist, or wasn't previously accessible).

The `current` status object may be a `StatStruct`, an `EOFError` (if the wait is canceled by closing this object), or some other `Exception` subtype (if the `stat` operation failed: for example, if the path is removed). Note that `stat` value may be outdated if the file has changed again multiple times.

Using `FileMonitor` for this operation is preferred, since it is more reliable and efficient, although in some situations it may not be available.

`FileWatching.FDWatcher` – Type.

```
FDWatcher(fd::Union{RawFD,WindowsRawSocket}, readable::Bool, writable::Bool)
```

Monitor a file descriptor `fd` for changes in the read or write availability.

The keyword arguments determine which of read and/or write status should be monitored; at least one of them must be set to `true`.

The returned value is an object with boolean fields `readable`, `writable`, and `timedout`, giving the result of the polling.

This acts like a level-set event, so calling `wait` blocks until one of those conditions is met, but then continues to return without blocking until the condition is cleared (either there is no more to read, or no more space in the write buffer, or both).

Warning

You must call `close` manually, when finished with this object, before the fd argument is closed. Failure to do so risks serious crashes.

Chapter 76

Pidfile

A simple utility tool for creating advisory pidfiles (lock files).

76.1 Primary Functions

`FileWatching.Pidfile.mkpidlock` – Function.

```
mkpidlock([f::Function], at::String, [pid::Cint]; kwopts...)
mkpidlock(at::String, proc::Process; kwopts...)
```

Create a pidfile lock for the path "at" for the current process or the process identified by pid or proc. Can take a function to execute once locked, for usage in do blocks, after which the lock will be automatically closed. If the lock fails and `wait` is false, then an error is thrown.

The lock will be released by either `close`, a `finalizer`, or shortly after `proc` exits. Make sure the return value is live through the end of the critical section of your program, so the `finalizer` does not reclaim it early.

Optional keyword arguments:

- `mode`: file access mode (modified by the process umask). Defaults to world-readable.
- `poll_interval`: Specify the maximum time to between attempts (if `watch_file` doesn't work)
- `stale_age`: Delete an existing pidfile (ignoring the lock) if it is older than this many seconds, based on its mtime. The file won't be deleted until 5x longer than this if the pid in the file appears that it may be valid. Or 25x longer if `refresh` is overridden to 0 to disable lock refreshing. By default this is disabled (`stale_age` = 0), but a typical recommended value would be about 3-5x an estimated normal completion time.
- `refresh`: Keeps a lock from becoming stale by updating the mtime every interval of time that passes. By default, this is set to `stale_age/2`, which is the recommended value.
- `wait`: If true, block until we get the lock, if false, raise error if lock fails.

`FileWatching.Pidfile.trymkpidlock` – Function.

```
trymkpidlock([f::Function], at::String, [pid::Cint]; kwopts...)
trymkpidlock(at::String, proc::Process; kwopts...)
```

Like `mkpidlock` except returns `false` instead of waiting if the file is already locked.

Julia 1.10

This function requires at least Julia 1.10.

`Base.close` – Method.

```
close(lock::LockMonitor)
```

Release a pidfile lock.

76.2 Helper Functions

`FileWatching.Pidfile.open_exclusive` – Function.

```
open_exclusive(path::String; mode, poll_interval, wait, stale_age, refresh) :: File
```

Create a new a file for read-write advisory-exclusive access. If `wait` is `false` then error out if the lock files exist otherwise block until we get the lock.

For a description of the keyword arguments, see `mkpidlock`.

`FileWatching.Pidfile.tryopen_exclusive` – Function.

```
tryopen_exclusive(path::String, mode::Integer = 0o444) :: Union{Void, File}
```

Try to create a new file for read-write advisory-exclusive access, return nothing if it already exists.

`FileWatching.Pidfile.write_pidfile` – Function.

```
write_pidfile(io, pid)
```

Write our pidfile format to an open IO descriptor.

`FileWatching.Pidfile.parse_pidfile` – Function.

```
parse_pidfile(file::Union{IO, String}) => (pid, hostname, age)
```

Attempt to parse our pidfile format, replaced an element with (0, "", 0.0), respectively, for any read that failed.

`FileWatching.Pidfile.stale_pidfile` – Function.

```
stale_pidfile(path::String, stale_age::Real, refresh::Real) :: Bool
```

Helper function for `open_exclusive` for deciding if a pidfile is stale.

`FileWatching.Pidfile.isvalidpid` – Function.

```
isvalidpid(hostname::String, pid::Cuint) :: Bool
```

Attempt to conservatively estimate whether pid is a valid process id.

`Base.Filesystem.touch` – Method.

```
Base.touch(::Pidfile.LockMonitor)
```

Update the `mtime` on the lock, to indicate it is still fresh.

See also the `refresh` keyword in the `mkpidlock` constructor.

Chapter 77

Future

The `Future` module implements future behavior of already existing functions, which will replace the current version in a future release of Julia.

`Future.copy!` – Function.

```
Future.copy!(dst, src) -> dst
```

Copy `src` into `dst`.

Julia 1.1

This function has moved to Base with Julia 1.1, consider using `copy!(dst, src)` instead. `Future.copy!` will be deprecated in the future.

`Future.randjump` – Function.

```
randjump(r::MersenneTwister, steps::Integer)::MersenneTwister
```

Create an initialized `MersenneTwister` object, whose state is moved forward (without generating numbers) from `r` by `steps` steps. One such step corresponds to the generation of two `Float64` numbers. For each different value of `steps`, a large polynomial has to be generated internally. One is already pre-computed for `steps=big(10)^20`.

Chapter 78

Interactive Utilities

The `InteractiveUtils` module provides utilities for interactive use of Julia, such as code introspection and clipboard access. It is intended for interactive work and is loaded automatically in interactive mode.

`Base.Docs.apropos` – Function.

```
apropos([io::IO=stdout], pattern::Union{AbstractString,Regex})
```

Search available docstrings for entries containing `pattern`.

When `pattern` is a string, case is ignored. Results are printed to `io`.

`apropos` can be called from the help mode in the REPL by wrapping the query in double quotes:

```
help?> "pattern"
```

Julia 1.11

In Julia 1.11 and newer, `apropos` requires that the REPL stdlib is loaded.

source

`InteractiveUtils.varinfo` – Function.

```
varinfo(m::Module=Main, pattern::Regex=r""; all=false, imported=false, recursive=false,
↪  sortby::Symbol=:name, minsize::Int=0)
```

Return a markdown table giving information about public global variables in a module, optionally restricted to those matching `pattern`.

The memory consumption estimate is an approximate lower bound on the size of the internal structure of the object.

- `all` : also list non-public objects defined in the module, deprecated objects, and compiler-generated objects.
- `imported` : also list objects explicitly imported from other modules.
- `recursive` : recursively include objects in sub-modules, observing the same settings in each.
- `sortby` : the column to sort results by. Options are `:name` (default), `:size`, and `:summary`.

- `minsize` : only includes objects with size at least `minsize` bytes. Defaults to `0`.

The output of `varinfo` is intended for display purposes only. See also `names` to get an array of symbols defined in a module, which is suitable for more general manipulations.

`InteractiveUtils.versioninfo` – Function.

```
versioninfo(io::IO=stdout; verbose::Bool=false)
```

Print information about the version of Julia in use. The output is controlled with boolean keyword arguments:

- `verbose`: print all additional information

Warning

The output of this function may contain sensitive information. Before sharing the output, please review the output and remove any data that should not be shared publicly.

See also: `VERSION`.

`InteractiveUtils.methodswith` – Function.

```
methodswith(typ[, module or function]; supertypes::Bool=false)
```

Return an array of methods with an argument of type `typ`.

The optional second argument restricts the search to a particular module or function (the default is all top-level modules).

If keyword `supertypes` is `true`, also return arguments with a parent type of `typ`, excluding type `Any`.

See also: `methods`.

`InteractiveUtils.subtypes` – Function.

```
subtypes(T::DataType)
```

Return a list of immediate subtypes of DataType T. Note that all currently loaded subtypes are included, including those not visible in the current module.

See also `supertype`, `supertypes`, `methodswith`.

Examples

```
julia> subtypes(Integer)
3-element Vector{Any}:
 Bool
 Signed
 Unsigned
```

`InteractiveUtils.supertypes` – Function.

```
supertypes(T::Type)
```

Return a tuple `(T, ..., Any)` of `T` and all its supertypes, as determined by successive calls to the `supertype` function, listed in order of `<:` and terminated by `Any`.

See also `subtypes`.

Examples

```
julia> supertypes(Int)
(Int64, Signed, Integer, Real, Number, Any)
```

`InteractiveUtils.edit` – Method.

```
edit(path::AbstractString, line::Integer=0, column::Integer=0)
```

Edit a file or directory optionally providing a line number to edit the file at. Return to the `julia` prompt when you quit the editor. The editor can be changed by setting `JULIA_EDITOR`, `VISUAL` or `EDITOR` as an environment variable.

Julia 1.9

The `column` argument requires at least Julia 1.9.

See also `InteractiveUtils.define_editor`.

`InteractiveUtils.edit` – Method.

```
edit(function, [types])
edit(module)
```

Edit the definition of a function, optionally specifying a tuple of types to indicate which method to edit. For modules, open the main source file. The module needs to be loaded with `using` or `import` first.

Julia 1.1

`edit` on modules requires at least Julia 1.1.

To ensure that the file can be opened at the given line, you may need to call `InteractiveUtils.define_editor` first.

`InteractiveUtils.@edit` – Macro.

```
@edit
```

Evaluates the arguments to the function or macro call, determines their types, and calls the `edit` function on the resulting expression.

See also: `@less`, `@which`.

`InteractiveUtils.define_editor` – Function.

```
define_editor(fn, pattern; wait=false)
```

Define a new editor matching `pattern` that can be used to open a file (possibly at a given line number) using `fn`.

The `fn` argument is a function that determines how to open a file with the given editor. It should take four arguments, as follows:

- `cmd` - a base command object for the editor
- `path` - the path to the source file to open
- `line` - the line number to open the editor at
- `column` - the column number to open the editor at

Editors which cannot open to a specific line with a command or a specific column may ignore the `line` and/or `column` argument. The `fn` callback must return either an appropriate `Cmd` object to open a file or `nothing` to indicate that they cannot edit this file. Use `nothing` to indicate that this editor is not appropriate for the current environment and another editor should be attempted. It is possible to add more general editing hooks that need not spawn external commands by pushing a callback directly to the vector `EDITOR_CALLBACKS`.

The `pattern` argument is a string, regular expression, or an array of strings and regular expressions. For the `fn` to be called, one of the patterns must match the value of `EDITOR`, `VISUAL` or `JULIA_EDITOR`. For strings, the string must equal the `basename` of the first word of the editor command, with its extension, if any, removed. E.g. "vi" doesn't match "vim -g" but matches "/usr/bin/vi -m"; it also matches `vi.exe`. If `pattern` is a regex it is matched against all of the editor command as a shell-escaped string. An array pattern matches if any of its items match. If multiple editors match, the one added most recently is used.

By default julia does not wait for the editor to close, running it in the background. However, if the editor is terminal based, you will probably want to set `wait=true` and julia will wait for the editor to close before resuming.

If one of the editor environment variables is set, but no editor entry matches it, the default editor entry is invoked:

```
(cmd, path, line, column) -> `$cmd $path`
```

Note that many editors are already defined. All of the following commands should already work:

- emacs
- emacsclient
- vim
- nvim
- nano
- micro
- kak
- helix
- textmate
- mate
- kate
- subl
- atom
- notepad++

- Visual Studio Code
- open
- pycharm
- bbedit

Examples

The following defines the usage of terminal-based `emacs`:

```
define_editor(
    r"\bemacs\b.*\s(-nw|--no-window-system)\b", wait=true) do cmd, path, line
    `$cmd +$line $path`
end
```

Julia 1.4

`define_editor` was introduced in Julia 1.4.

`InteractiveUtils.less` – Method.

```
less(file::AbstractString, [line::Integer])
```

Show a file using the default pager, optionally providing a starting line number. Returns to the `julia` prompt when you quit the pager.

`InteractiveUtils.less` – Method.

```
less(function, [types])
```

Show the definition of a function using the default pager, optionally specifying a tuple of types to indicate which method to see.

`InteractiveUtils.@less` – Macro.

```
@less
```

Evaluates the arguments to the function or macro call, determines their types, and calls the `less` function on the resulting expression.

See also: `@edit`, `@which`, `@code_lowered`.

`InteractiveUtils.@which` – Macro.

```
@which
```

Applied to a function or macro call, it evaluates the arguments to the specified call, and returns the `Method` object for the method that would be called for those arguments. Applied to a variable, it returns the module in which the variable was bound. It calls out to the `which` function.

See also: `@less`, `@edit`.

`InteractiveUtils.@functionloc` – Macro.

```
@functionloc
```

Applied to a function or macro call, it evaluates the arguments to the specified call, and returns a tuple `(filename,line)` giving the location for the method that would be called for those arguments. It calls out to the `functionloc` function.

`InteractiveUtils.@code_lowered` – Macro.

```
@code_lowered
```

Evaluates the arguments to the function or macro call, determines their types, and calls `code_lowered` on the resulting expression.

See also: `code_lowered`, `@code_warntype`, `@code_typed`, `@code_llvm`, `@code_native`.

`InteractiveUtils.@code_typed` – Macro.

```
@code_typed
```

Evaluates the arguments to the function or macro call, determines their types, and calls `code_typed` on the resulting expression. Use the optional argument `optimize` with

```
@code_typed optimize=true foo(x)
```

to control whether additional optimizations, such as inlining, are also applied.

See also: `code_typed`, `@code_warntype`, `@code_lowered`, `@code_llvm`, `@code_native`.

`InteractiveUtils.code_warntype` – Function.

```
code_warntype([io::IO], f, types; debuginfo=:default)
```

Prints lowered and type-inferred ASTs for the methods matching the given generic function and type signature to `io` which defaults to `stdout`. The ASTs are annotated in such a way as to cause non-concrete types which may be problematic for performance to be emphasized (if color is available, displayed in red). This serves as a warning of potential type instability.

Not all non-concrete types are particularly problematic for performance, and the performance characteristics of a particular type is an implementation detail of the compiler. `code_warntype` will err on the side of coloring types red if they might be a performance concern, so some types may be colored red even if they do not impact performance. Small unions of concrete types are usually not a concern, so these are highlighted in yellow.

Keyword argument `debuginfo` may be one of `:source`, `:none` or `:default`, to specify the verbosity of code comments. Unless the user changes `Base.IRShow.default_debuginfo[]`, the value `:default` is equivalent to `:source`.

See the `@code_warntype` section in the Performance Tips page of the manual for more information.

See also: `@code_warntype`, `code_typed`, `code_lowered`, `code_llvm`, `code_native`.

`InteractiveUtils.@code_warntype` – Macro.

```
@code_warntype
```

Evaluates the arguments to the function or macro call, determines their types, and calls code_warntype on the resulting expression.

See also: code_warntype, @code_typed, @code_lowered, @code_llvm, @code_native.

InteractiveUtils.code_llvm – Function.

```
code_llvm([io=stdout,], f, types; raw=false, dump_module=false, optimize=true,
↪  debuginfo=:default)
```

Prints the LLVM bitcodes generated for running the method matching the given generic function and type signature to io.

If the optimize keyword is unset, the code will be shown before LLVM optimizations. All metadata and dbg.* calls are removed from the printed bitcode. For the full IR, set the raw keyword to true. To dump the entire module that encapsulates the function (with declarations), set the dump_module keyword to true. Keyword argument debuginfo may be one of source (default) or none, to specify the verbosity of code comments.

See also: @code_llvm, code_warntype, code_typed, code_lowered, code_native.

InteractiveUtils.@code_llvm – Macro.

```
@code_llvm
```

Evaluates the arguments to the function or macro call, determines their types, and calls code_llvm on the resulting expression. Set the optional keyword arguments raw, dump_module, debuginfo, optimize by putting them and their value before the function call, like this:

```
@code_llvm raw=true dump_module=true debuginfo=:default f(x)
@code_llvm optimize=false f(x)
```

optimize controls whether additional optimizations, such as inlining, are also applied. raw makes all metadata and dbg.* calls visible. debuginfo may be one of :source (default) or :none, to specify the verbosity of code comments. dump_module prints the entire module that encapsulates the function.

See also: code_llvm, @code_warntype, @code_typed, @code_lowered, @code_native.

InteractiveUtils.code_native – Function.

```
code_native([io=stdout,], f, types; syntax=:intel, debuginfo=:default, binary=false,
↪  dump_module=true, raw=false)
```

Prints the native assembly instructions generated for running the method matching the given generic function and type signature to io.

- Set assembly syntax by setting syntax to :intel (default) for intel syntax or :att for AT&T syntax.
- Specify verbosity of code comments by setting debuginfo to :source (equivalently, :default) or :none.
- If binary is true, also print the binary machine code for each instruction precedented by an abbreviated address.
- If dump_module is false, do not print metadata such as rodata or directives.

- If `raw` is `false` (default), uninteresting instructions (like the safepoint function prologue) are elided.

See also: `@code_native`, `code_warntype`, `code_typed`, `code_lowered`, `code_llvm`.

`InteractiveUtils.@code_native` – Macro.

```
@code_native
```

Evaluates the arguments to the function or macro call, determines their types, and calls `code_native` on the resulting expression.

Set any of the optional keyword arguments `syntax`, `debuginfo`, `binary` or `dump_module` by putting it before the function call, like this:

```
@code_native syntax=:intel debuginfo=:default binary=true dump_module=false f(x)
```

- Set assembly syntax by setting `syntax` to `:intel` (default) for Intel syntax or `:att` for AT&T syntax.
- Specify verbosity of code comments by setting `debuginfo` to `:source` (default) or `:none`.
- If `binary` is `true`, also print the binary machine code for each instruction precedented by an abbreviated address.
- If `dump_module` is `false`, do not print metadata such as rodata or directives.

See also: `code_native`, `@code_warntype`, `@code_typed`, `@code_lowered`, `@code_llvm`.

`Base.@time_imports` – Macro.

```
@time_imports
```

A macro to execute an expression and produce a report of any time spent importing packages and their dependencies. Any compilation time will be reported as a percentage, and how much of which was recompilation, if any.

One line is printed per package or package extension. The duration shown is the time to import that package itself, not including the time to load any of its dependencies.

On Julia 1.9+ package extensions will show as Parent → Extension.

Note

During the load process a package sequentially imports all of its dependencies, not just its direct dependencies.

```
julia> @time_imports using CSV
     50.7 ms  Parsers 17.52% compilation time
      0.2 ms  DataValueInterfaces
      1.6 ms  DataAPI
      0.1 ms  IteratorInterfaceExtensions
      0.1 ms  TableTraits
     17.5 ms  Tables
     26.8 ms  PooledArrays
```

```
   193.7 ms  SentinelArrays 75.12% compilation time
     8.6 ms  InlineStrings
    20.3 ms  WeakRefStrings
     2.0 ms  TranscodingStreams
     1.4 ms  Zlib_jll
     1.8 ms  CodecZlib
     0.8 ms  Compat
    13.1 ms  FilePathsBase 28.39% compilation time
  1681.2 ms  CSV 92.40% compilation time
```

Julia 1.8

This macro requires at least Julia 1.8

`Base.@trace_compile` – Macro.

```
@trace_compile
```

A macro to execute an expression and show any methods that were compiled (or recompiled in yellow), like the julia args `--trace-compile=stderr --trace-compile-timing` but specifically for a call.

```
julia> @trace_compile rand(2,2) * rand(2,2)
#=   39.1 ms =# precompile(Tuple{typeof(Base.rand), Int64, Int64})
#=  102.0 ms =# precompile(Tuple{typeof(Base.:(*)), Array{Float64, 2}, Array{Float64, 2}})
2×2 Matrix{Float64}:
 0.421704  0.864841
 0.211262  0.444366
```

Julia 1.12

This macro requires at least Julia 1.12

`Base.@trace_dispatch` – Macro.

```
@trace_dispatch
```

A macro to execute an expression and report methods that were compiled via dynamic dispatch, like the julia arg `--trace-dispatch=stderr` but specifically for a call.

Julia 1.12

This macro requires at least Julia 1.12

`InteractiveUtils.clipboard` – Function.

```
clipboard(x)
```

Send a printed form of x to the operating system clipboard ("copy").

```
clipboard()::String
```

Return a string with the contents of the operating system clipboard ("paste").

Chapter 79

Julia Syntax Highlighting

The `JuliaSyntaxHighlighting` library serves as a small convenience package to syntax highlight Julia code using `JuliaSyntax` and `StyledStrings`.

It is intended for use across the standard library, and the wider ecosystem.

79.1 Functions

`JuliaSyntaxHighlighting.highlight` – Function.

```
highlight(content::Union{AbstractString, IO},
          ast::JuliaSyntax.GreenNode = <parsed content>;
          syntax_errors::Bool = false) -> AnnotatedString{String}
```

Apply syntax highlighting to `content` using `JuliaSyntax`.

By default, `JuliaSyntax.parseall` is used to generate to `ast` with the `ignore_errors` keyword argument set to `true`. Alternatively, one may provide a pre-generated `ast`.

When `syntax_errors` is set, the `julia_error` face is applied to detected syntax errors.

Warning

Note that the particular faces used by `JuliaSyntax`, and the way they are applied, is subject to change.

Examples

```
julia> JuliaSyntaxHighlighting.highlight("sum(1:8)")
"sum(1:8)"

julia> JuliaSyntaxHighlighting.highlight("sum(1:8)") |> Base.annotations
6-element Vector{@NamedTuple{region::UnitRange{Int64}, label::Symbol, value}}:
 @NamedTuple{region::UnitRange{Int64}, label::Symbol, value}((1:3, :face, :julia_funcall))
 @NamedTuple{region::UnitRange{Int64}, label::Symbol, value}((4:4, :face,
 ↪  :julia_rainbow_paren_1))
 @NamedTuple{region::UnitRange{Int64}, label::Symbol, value}((5:5, :face, :julia_number))
 @NamedTuple{region::UnitRange{Int64}, label::Symbol, value}((6:6, :face, :julia_operator))
 @NamedTuple{region::UnitRange{Int64}, label::Symbol, value}((7:7, :face, :julia_number))
```

```
@NamedTuple{region::UnitRange{Int64}, label::Symbol, value}((8:8, :face,
↪  :julia_rainbow_paren_1))
```

`JuliaSyntaxHighlighting.highlight!` – Function.

```
highlight!(content::Union{AnnotatedString, SubString{AnnotatedString}},
           ast::JuliaSyntax.GreenNode = <parsed content>;
           syntax_errors::Bool = false) -> content
```

Modify `content` by applying syntax highlighting using `JuliaSyntax`.

By default, `JuliaSyntax.parseall` is used to generate to `ast` with the `ignore_errors` keyword argument set to `true`. Alternatively, one may provide a pre-generated `ast`.

When `syntax_errors` is set, the `julia_error` face is applied to detected syntax errors.

Warning

Note that the particular faces used by `JuliaSyntax`, and the way they are applied, is subject to change.

Examples

```
julia> str = Base.AnnotatedString("sum(1:8)")
"sum(1:8)"

julia> JuliaSyntaxHighlighting.highlight!(str)
"sum(1:8)"

julia> Base.annotations(str)
6-element Vector{@NamedTuple{region::UnitRange{Int64}, label::Symbol, value}}:
 @NamedTuple{region::UnitRange{Int64}, label::Symbol, value}((1:3, :face, :julia_funcall))
 @NamedTuple{region::UnitRange{Int64}, label::Symbol, value}((4:4, :face,
 ↪  :julia_rainbow_paren_1))
 @NamedTuple{region::UnitRange{Int64}, label::Symbol, value}((5:5, :face, :julia_number))
 @NamedTuple{region::UnitRange{Int64}, label::Symbol, value}((6:6, :face, :julia_operator))
 @NamedTuple{region::UnitRange{Int64}, label::Symbol, value}((7:7, :face, :julia_number))
 @NamedTuple{region::UnitRange{Int64}, label::Symbol, value}((8:8, :face,
 ↪  :julia_rainbow_paren_1))
```

79.2 Faces

The `highlight`/`highlight!` methods work by applying custom faces to Julia code. As part of the standard library, these faces use privileged face names, of the form `julia_*`. These can be re-used in other packages, and customised with `faces.toml` configuration.

Unstable faces

The particular faces used by `JuliaSyntaxHighlighting` are liable to change without warning in point releases. As the syntax highlighting rules are refined over time, changes should become less and less frequent though.

The current set of faces, and their default values are as follows:

- `julia_macro`: magenta
- `julia_symbol`: magenta
- `julia_singleton_identifier`: inherits from `julia_symbol`
- `julia_type`: yellow
- `julia_typedec`: bright blue
- `julia_comment`: grey
- `julia_string`: green
- `julia_regex`: inherits from `julia_string`
- `julia_backslash_literal`: magenta, inherits from `julia_string`
- `julia_string_delim`: bright green
- `julia_cmdstring`: inherits from `julia_string`
- `julia_char`: inherits from `julia_string`
- `julia_char_delim`: inherits from `julia_string_delim`
- `julia_number`: bright magenta
- `julia_bool`: inherits from `julia_number`
- `julia_funcall`: cyan
- `julia_funcdef`: cyan
- `julia_broadcast`: bright blue, bold
- `julia_builtin`: bright blue
- `julia_operator`: blue
- `julia_comparator`: inherits from `julia_operator`
- `julia_assignment`: bright red
- `julia_keyword`: red
- `julia_parentheses`: unstyled
- `julia_unpaired_parentheses`: inherit from `julia_error` and `julia_parentheses`
- `julia_error`: red background
- `julia_rainbow_paren_1`: bright green, inherits from `julia_parentheses`
- `julia_rainbow_paren_2`: bright blue, inherits from `julia_parentheses`
- `julia_rainbow_paren_3`: bright red, inherits from `julia_parentheses`
- `julia_rainbow_paren_4`: inherits from `julia_rainbow_paren_1`

- `julia_rainbow_paren_5`: inherits from `julia_rainbow_paren_2`
- `julia_rainbow_paren_6`: inherits from `julia_rainbow_paren_3`
- `julia_rainbow_bracket_1`: blue, inherits from `julia_parentheses`
- `julia_rainbow_bracket_2`: bright*magenta, inherits from 'julia*`parentheses`'
- `julia_rainbow_bracket_3`: inherits from `julia_rainbow_bracket_1`
- `julia_rainbow_bracket_4`: inherits from `julia_rainbow_bracket_2`
- `julia_rainbow_bracket_5`: inherits from `julia_rainbow_bracket_1`
- `julia_rainbow_bracket_6`: inherits from `julia_rainbow_bracket_2`
- `julia_rainbow_curly_1`: bright yellow, inherits from `julia_parentheses`
- `julia_rainbow_curly_2`: yellow, inherits from `julia_parentheses`
- `julia_rainbow_curly_3`: inherits from `julia_rainbow_curly_1`
- `julia_rainbow_curly_4`: inherits from `julia_rainbow_curly_2`
- `julia_rainbow_curly_5`: inherits from `julia_rainbow_curly_1`
- `julia_rainbow_curly_6`: inherits from `julia_rainbow_curly_2`

Chapter 80

Lazy Artifacts

In order for a package to download artifacts lazily, `LazyArtifacts` must be explicitly listed as a dependency of that package.

For further information on artifacts, see Artifacts.

Chapter 81

LibCURL

This is a simple Julia wrapper around http://curl.haxx.se/libcurl/ generated using Clang.jl. Please see the libcurl API documentation for help on how to use this package.

Chapter 82

LibGit2

The LibGit2 module provides bindings to libgit2, a portable C library that implements core functionality for the Git version control system. These bindings are currently used to power Julia's package manager. It is expected that this module will eventually be moved into a separate package.

Functionality

Some of this documentation assumes some prior knowledge of the libgit2 API. For more information on some of the objects and methods referenced here, consult the upstream libgit2 API reference.

`LibGit2.Buffer` – Type.

```
LibGit2.Buffer
```

A data buffer for exporting data from libgit2. Matches the `git_buf` struct.

When fetching data from LibGit2, a typical usage would look like:

```
buf_ref = Ref(Buffer())
@check ccall(..., (Ptr{Buffer},), buf_ref)
# operation on buf_ref
free(buf_ref)
```

In particular, note that `LibGit2.free` should be called afterward on the `Ref` object.

`LibGit2.CheckoutOptions` – Type.

```
LibGit2.CheckoutOptions
```

Matches the `git_checkout_options` struct.

The fields represent:

- `version`: version of the struct in use, in case this changes later. For now, always 1.
- `checkout_strategy`: determine how to handle conflicts and whether to force the checkout/recreate missing files.
- `disable_filters`: if nonzero, do not apply filters like CLRF (to convert file newlines between UNIX and DOS).

- `dir_mode`: read/write/access mode for any directories involved in the checkout. Default is `0755`.
- `file_mode`: read/write/access mode for any files involved in the checkout. Default is `0755` or `0644`, depending on the blob.
- `file_open_flags`: bitflags used to open any files during the checkout.
- `notify_flags`: Flags for what sort of conflicts the user should be notified about.
- `notify_cb`: An optional callback function to notify the user if a checkout conflict occurs. If this function returns a non-zero value, the checkout will be cancelled.
- `notify_payload`: Payload for the notify callback function.
- `progress_cb`: An optional callback function to display checkout progress.
- `progress_payload`: Payload for the progress callback.
- `paths`: If not empty, describes which paths to search during the checkout. If empty, the checkout will occur over all files in the repository.
- `baseline`: Expected content of the `workdir`, captured in a (pointer to a) `GitTree`. Defaults to the state of the tree at HEAD.
- `baseline_index`: Expected content of the `workdir`, captured in a (pointer to a) `GitIndex`. Defaults to the state of the index at HEAD.
- `target_directory`: If not empty, checkout to this directory instead of the `workdir`.
- `ancestor_label`: In case of conflicts, the name of the common ancestor side.
- `our_label`: In case of conflicts, the name of "our" side.
- `their_label`: In case of conflicts, the name of "their" side.
- `perfdata_cb`: An optional callback function to display performance data.
- `perfdata_payload`: Payload for the performance callback.

`LibGit2.CloneOptions` – Type.

```
LibGit2.CloneOptions
```

Matches the `git_clone_options` struct.

The fields represent:

- `version`: version of the struct in use, in case this changes later. For now, always `1`.
- `checkout_opts`: The options for performing the checkout of the remote as part of the clone.
- `fetch_opts`: The options for performing the pre-checkout fetch of the remote as part of the clone.
- `bare`: If `0`, clone the full remote repository. If non-zero, perform a bare clone, in which there is no local copy of the source files in the repository and the `gitdir` and `workdir` are the same.
- `localclone`: Flag whether to clone a local object database or do a fetch. The default is to let git decide. It will not use the git-aware transport for a local clone, but will use it for URLs which begin with `file://`.
- `checkout_branch`: The name of the branch to checkout. If an empty string, the default branch of the remote will be checked out.
- `repository_cb`: An optional callback which will be used to create the *new* repository into which the clone is made.

- `repository_cb_payload`: The payload for the repository callback.
- `remote_cb`: An optional callback used to create the `GitRemote` before making the clone from it.
- `remote_cb_payload`: The payload for the remote callback.

`LibGit2.DescribeOptions` – Type.

```
LibGit2.DescribeOptions
```

Matches the `git_describe_options` struct.

The fields represent:

- `version`: version of the struct in use, in case this changes later. For now, always 1.
- `max_candidates_tags`: consider this many most recent tags in `refs/tags` to describe a commit. Defaults to 10 (so that the 10 most recent tags would be examined to see if they describe a commit).
- `describe_strategy`: whether to consider all entries in `refs/tags` (equivalent to `git-describe --tags`) or all entries in `refs/` (equivalent to `git-describe --all`). The default is to only show annotated tags. If `Consts.DESCRIBE_TAGS` is passed, all tags, annotated or not, will be considered. If `Consts.DESCRIBE_ALL` is passed, any ref in `refs/` will be considered.
- `pattern`: only consider tags which match `pattern`. Supports glob expansion.
- `only_follow_first_parent`: when finding the distance from a matching reference to the described object, only consider the distance from the first parent.
- `show_commit_oid_as_fallback`: if no matching reference can be found which describes a commit, show the commit's `GitHash` instead of throwing an error (the default behavior).

`LibGit2.DescribeFormatOptions` – Type.

```
LibGit2.DescribeFormatOptions
```

Matches the `git_describe_format_options` struct.

The fields represent:

- `version`: version of the struct in use, in case this changes later. For now, always 1.
- `abbreviated_size`: lower bound on the size of the abbreviated `GitHash` to use, defaulting to 7.
- `always_use_long_format`: set to 1 to use the long format for strings even if a short format can be used.
- `dirty_suffix`: if set, this will be appended to the end of the description string if the `workdir` is dirty.

`LibGit2.DiffDelta` – Type.

```
LibGit2.DiffDelta
```

Description of changes to one entry. Matches the `git_diff_delta` struct.

The fields represent:

- `status`: One of `Consts.DELTA_STATUS`, indicating whether the file has been added/modified/deleted.
- `flags`: Flags for the delta and the objects on each side. Determines whether to treat the file(s) as binary/text, whether they exist on each side of the diff, and whether the object ids are known to be correct.
- `similarity`: Used to indicate if a file has been renamed or copied.
- `nfiles`: The number of files in the delta (for instance, if the delta was run on a submodule commit id, it may contain more than one file).
- `old_file`: A `DiffFile` containing information about the file(s) before the changes.
- `new_file`: A `DiffFile` containing information about the file(s) after the changes.

`LibGit2.DiffFile` – Type.

```
LibGit2.DiffFile
```

Description of one side of a delta. Matches the `git_diff_file` struct.

The fields represent:

- `id`: the `GitHash` of the item in the diff. If the item is empty on this side of the diff (for instance, if the diff is of the removal of a file), this will be `GitHash(0)`.
- `path`: a `NULL` terminated path to the item relative to the working directory of the repository.
- `size`: the size of the item in bytes.
- `flags`: a combination of the `git_diff_flag_t` flags. The `i`th bit of this integer sets the `i`th flag.
- `mode`: the `stat` mode for the item.
- `id_abbrev`: only present in LibGit2 versions newer than or equal to `0.25.0`. The length of the `id` field when converted using `string`. Usually equal to `OID_HEXSZ` (40).

`LibGit2.DiffOptionsStruct` – Type.

```
LibGit2.DiffOptionsStruct
```

Matches the `git_diff_options` struct.

The fields represent:

- `version`: version of the struct in use, in case this changes later. For now, always 1.
- `flags`: flags controlling which files will appear in the diff. Defaults to `DIFF_NORMAL`.
- `ignore_submodules`: whether to look at files in submodules or not. Defaults to `SUBMODULE_IGNORE_UNSPECIFIED`, which means the submodule's configuration will control whether it appears in the diff or not.
- `pathspec`: path to files to include in the diff. Default is to use all files in the repository.
- `notify_cb`: optional callback which will notify the user of changes to the diff as file deltas are added to it.
- `progress_cb`: optional callback which will display diff progress. Only relevant on libgit2 versions at least as new as 0.24.0.
- `payload`: the payload to pass to `notify_cb` and `progress_cb`.

- `context_lines`: the number of *unchanged* lines used to define the edges of a hunk. This is also the number of lines which will be shown before/after a hunk to provide context. Default is 3.
- `interhunk_lines`: the maximum number of *unchanged* lines *between* two separate hunks allowed before the hunks will be combined. Default is 0.
- `id_abbrev`: sets the length of the abbreviated `GitHash` to print. Default is 7.
- `max_size`: the maximum file size of a blob. Above this size, it will be treated as a binary blob. The default is 512 MB.
- `old_prefix`: the virtual file directory in which to place old files on one side of the diff. Default is `"a"`.
- `new_prefix`: the virtual file directory in which to place new files on one side of the diff. Default is `"b"`.

`LibGit2.FetchHead` – Type.

```
LibGit2.FetchHead
```

Contains the information about HEAD during a fetch, including the name and URL of the branch fetched from, the oid of the HEAD, and whether the fetched HEAD has been merged locally.

The fields represent:

- `name`: The name in the local reference database of the fetch head, for example, `"refs/heads/master"`.
- `url`: The URL of the fetch head.
- `oid`: The `GitHash` of the tip of the fetch head.
- `ismerge`: Boolean flag indicating whether the changes at the remote have been merged into the local copy yet or not. If `true`, the local copy is up to date with the remote fetch head.

`LibGit2.FetchOptions` – Type.

```
LibGit2.FetchOptions
```

Matches the `git_fetch_options` struct.

The fields represent:

- `version`: version of the struct in use, in case this changes later. For now, always 1.
- `callbacks`: remote callbacks to use during the fetch.
- `prune`: whether to perform a prune after the fetch or not. The default is to use the setting from the `GitConfig`.
- `update_fetchhead`: whether to update the `FetchHead` after the fetch. The default is to perform the update, which is the normal git behavior.
- `download_tags`: whether to download tags present at the remote or not. The default is to request the tags for objects which are being downloaded anyway from the server.
- `proxy_opts`: options for connecting to the remote through a proxy. See `ProxyOptions`. Only present on libgit2 versions newer than or equal to 0.25.0.
- `custom_headers`: any extra headers needed for the fetch. Only present on libgit2 versions newer than or equal to 0.24.0.

LibGit2.GitAnnotated – Type.

```
GitAnnotated(repo::GitRepo, commit_id::GitHash)
GitAnnotated(repo::GitRepo, ref::GitReference)
GitAnnotated(repo::GitRepo, fh::FetchHead)
GitAnnotated(repo::GitRepo, committish::AbstractString)
```

An annotated git commit carries with it information about how it was looked up and why, so that rebase or merge operations have more information about the context of the commit. Conflict files contain information about the source/target branches in the merge which are conflicting, for instance. An annotated commit can refer to the tip of a remote branch, for instance when a FetchHead is passed, or to a branch head described using GitReference.

LibGit2.GitBlame – Type.

```
GitBlame(repo::GitRepo, path::AbstractString; options::BlameOptions=BlameOptions())
```

Construct a GitBlame object for the file at path, using change information gleaned from the history of repo. The GitBlame object records who changed which chunks of the file when, and how. options controls how to separate the contents of the file and which commits to probe - see BlameOptions for more information.

LibGit2.GitBlob – Type.

```
GitBlob(repo::GitRepo, hash::AbstractGitHash)
GitBlob(repo::GitRepo, spec::AbstractString)
```

Return a GitBlob object from repo specified by hash/spec.

- hash is a full (GitHash) or partial (GitShortHash) hash.
- spec is a textual specification: see the git docs for a full list.

LibGit2.GitCommit – Type.

```
GitCommit(repo::GitRepo, hash::AbstractGitHash)
GitCommit(repo::GitRepo, spec::AbstractString)
```

Return a GitCommit object from repo specified by hash/spec.

- hash is a full (GitHash) or partial (GitShortHash) hash.
- spec is a textual specification: see the git docs for a full list.

LibGit2.GitConfig – Type.

```
GitConfig(path::AbstractString, level::Consts.GIT_CONFIG=Consts.CONFIG_LEVEL_APP,
↪  force::Bool=false)
```

Create a new GitConfig by loading configuration information from the file at path. See addfile for more information about the level, repo and force options.

```
GitConfig(repo::GitRepo)
```

Get the stored configuration for the git repository `repo`. If `repo` does not have a specific configuration file set, the default git configuration will be used.

```
GitConfig(level::Consts.GIT_CONFIG=Consts.CONFIG_LEVEL_DEFAULT)
```

Get the default git configuration by loading the global and system configuration files into a prioritized configuration. This can be used to access default configuration options outside a specific git repository.

`LibGit2.GitHash` – Type.

```
GitHash
```

A git object identifier, based on the sha-1 hash. It is a 20 byte string (40 hex digits) used to identify a `GitObject` in a repository.

`LibGit2.GitObject` – Type.

```
GitObject(repo::GitRepo, hash::AbstractGitHash)
GitObject(repo::GitRepo, spec::AbstractString)
```

Return the specified object (`GitCommit`, `GitBlob`, `GitTree` or `GitTag`) from `repo` specified by `hash`/`spec`.

- `hash` is a full (`GitHash`) or partial (`GitShortHash`) hash.
- `spec` is a textual specification: see the git docs for a full list.

`LibGit2.GitRemote` – Type.

```
GitRemote(repo::GitRepo, rmt_name::AbstractString, rmt_url::AbstractString)::GitRemote
```

Look up a remote git repository using its name and URL. Uses the default fetch refspec.

Examples

```
repo = LibGit2.init(repo_path)
remote = LibGit2.GitRemote(repo, "upstream", repo_url)
```

```
GitRemote(repo::GitRepo, rmt_name::AbstractString, rmt_url::AbstractString,
↪  fetch_spec::AbstractString)::GitRemote
```

Look up a remote git repository using the repository's name and URL, as well as specifications for how to fetch from the remote (e.g. which remote branch to fetch from).

Examples

```
repo = LibGit2.init(repo_path)
refspec = "+refs/heads/mybranch:refs/remotes/origin/mybranch"
remote = LibGit2.GitRemote(repo, "upstream", repo_url, refspec)
```

`LibGit2.GitRemoteAnon` – Function.

```
GitRemoteAnon(repo::GitRepo, url::AbstractString)::GitRemote
```

Look up a remote git repository using only its URL, not its name.

Examples

```
repo = LibGit2.init(repo_path)
remote = LibGit2.GitRemoteAnon(repo, repo_url)
```

`LibGit2.GitRepo` – Type.

```
LibGit2.GitRepo(path::AbstractString)
```

Open a git repository at `path`.

`LibGit2.GitRepoExt` – Function.

```
LibGit2.GitRepoExt(path::AbstractString, flags::Cuint = Cuint(Consts.REPOSITORY_OPEN_DEFAULT))
```

Open a git repository at `path` with extended controls (for instance, if the current user must be a member of a special access group to read `path`).

`LibGit2.GitRevWalker` – Type.

```
GitRevWalker(repo::GitRepo)
```

A `GitRevWalker` *walks* through the *revisions* (i.e. commits) of a git repository `repo`. It is a collection of the commits in the repository, and supports iteration and calls to `LibGit2.map` and `LibGit2.count` (for instance, `LibGit2.count` could be used to determine what percentage of commits in a repository were made by a certain author).

```
cnt = LibGit2.with(LibGit2.GitRevWalker(repo)) do walker
    LibGit2.count((oid,repo)->(oid == commit_oid1), walker, oid=commit_oid1,
    ↪  by=LibGit2.Consts.SORT_TIME)
end
```

Here, `LibGit2.count` finds the number of commits along the walk with a certain `GitHash`. Since the `GitHash` is unique to a commit, `cnt` will be 1.

`LibGit2.GitShortHash` – Type.

```
GitShortHash(hash::GitHash, len::Integer)
```

A shortened git object identifier, which can be used to identify a git object when it is unique, consisting of the initial `len` hexadecimal digits of `hash` (the remaining digits are ignored).

`LibGit2.GitSignature` – Type.

```
LibGit2.GitSignature
```

This is a Julia wrapper around a pointer to a `git_signature` object.

`LibGit2.GitStatus` – Type.

```
LibGit2.GitStatus(repo::GitRepo; status_opts=StatusOptions())
```

Collect information about the status of each file in the git repository `repo` (e.g. is the file modified, staged, etc.). `status_opts` can be used to set various options, for instance whether or not to look at untracked files or whether to include submodules or not. See `StatusOptions` for more information.

`LibGit2.GitTag` – Type.

```
GitTag(repo::GitRepo, hash::AbstractGitHash)
GitTag(repo::GitRepo, spec::AbstractString)
```

Return a `GitTag` object from `repo` specified by `hash`/`spec`.

- `hash` is a full (`GitHash`) or partial (`GitShortHash`) hash.
- `spec` is a textual specification: see the git docs for a full list.

`LibGit2.GitTree` – Type.

```
GitTree(repo::GitRepo, hash::AbstractGitHash)
GitTree(repo::GitRepo, spec::AbstractString)
```

Return a `GitTree` object from `repo` specified by `hash`/`spec`.

- `hash` is a full (`GitHash`) or partial (`GitShortHash`) hash.
- `spec` is a textual specification: see the git docs for a full list.

`LibGit2.IndexEntry` – Type.

```
LibGit2.IndexEntry
```

In-memory representation of a file entry in the index. Matches the `git_index_entry` struct.

`LibGit2.IndexTime` – Type.

```
LibGit2.IndexTime
```

Matches the `git_index_time` struct.

`LibGit2.BlameOptions` – Type.

```
LibGit2.BlameOptions
```

Matches the `git_blame_options` struct.

The fields represent:

- `version`: version of the struct in use, in case this changes later. For now, always 1.
- `flags`: one of `Consts.BLAME_NORMAL` or `Consts.BLAME_FIRST_PARENT` (the other blame flags are not yet implemented by libgit2).
- `min_match_characters`: the minimum number of *alphanumeric* characters which much change in a commit in order for the change to be associated with that commit. The default is 20. Only takes effect if one of the `Consts.BLAME_*_COPIES` flags are used, which libgit2 does not implement yet.

- `newest_commit`: the `GitHash` of the newest commit from which to look at changes.
- `oldest_commit`: the `GitHash` of the oldest commit from which to look at changes.
- `min_line`: the first line of the file from which to starting blaming. The default is `1`.
- `max_line`: the last line of the file to which to blame. The default is `0`, meaning the last line of the file.

`LibGit2.MergeOptions` – Type.

```
LibGit2.MergeOptions
```

Matches the `git_merge_options` struct.

The fields represent:

- `version`: version of the struct in use, in case this changes later. For now, always `1`.
- `flags`: an enum for flags describing merge behavior. Defined in `git_merge_flag_t`. The corresponding Julia enum is `GIT_MERGE` and has values:
  - `MERGE_FIND_RENAMES`: detect if a file has been renamed between the common ancestor and the "ours" or "theirs" side of the merge. Allows merges where a file has been renamed.
  - `MERGE_FAIL_ON_CONFLICT`: exit immediately if a conflict is found rather than trying to resolve it.
  - `MERGE_SKIP_REUC`: do not write the REUC extension on the index resulting from the merge.
  - `MERGE_NO_RECURSIVE`: if the commits being merged have multiple merge bases, use the first one, rather than trying to recursively merge the bases.
- `rename_threshold`: how similar two files must to consider one a rename of the other. This is an integer that sets the percentage similarity. The default is `50`.
- `target_limit`: the maximum number of files to compare with to look for renames. The default is `200`.
- `metric`: optional custom function to use to determine the similarity between two files for rename detection.
- `recursion_limit`: the upper limit on the number of merges of common ancestors to perform to try to build a new virtual merge base for the merge. The default is no limit. This field is only present on libgit2 versions newer than 0.24.0.
- `default_driver`: the merge driver to use if both sides have changed. This field is only present on libgit2 versions newer than 0.25.0.
- `file_favor`: how to handle conflicting file contents for the `text` driver.
  - `MERGE_FILE_FAVOR_NORMAL`: if both sides of the merge have changes to a section, make a note of the conflict in the index which `git checkout` will use to create a merge file, which the user can then reference to resolve the conflicts. This is the default.
  - `MERGE_FILE_FAVOR_OURS`: if both sides of the merge have changes to a section, use the version in the "ours" side of the merge in the index.
  - `MERGE_FILE_FAVOR_THEIRS`: if both sides of the merge have changes to a section, use the version in the "theirs" side of the merge in the index.
  - `MERGE_FILE_FAVOR_UNION`: if both sides of the merge have changes to a section, include each unique line from both sides in the file which is put into the index.

- `file_flags`: guidelines for merging files.

`LibGit2.ProxyOptions` – Type.

```
LibGit2.ProxyOptions
```

Options for connecting through a proxy.

Matches the `git_proxy_options` struct.

The fields represent:

- `version`: version of the struct in use, in case this changes later. For now, always 1.
- `proxytype`: an enum for the type of proxy to use. Defined in `git_proxy_t`. The corresponding Julia enum is `GIT_PROXY` and has values:
  - `PROXY_NONE`: do not attempt the connection through a proxy.
  - `PROXY_AUTO`: attempt to figure out the proxy configuration from the git configuration.
  - `PROXY_SPECIFIED`: connect using the URL given in the `url` field of this struct.

  Default is to auto-detect the proxy type.
- `url`: the URL of the proxy.
- `credential_cb`: a pointer to a callback function which will be called if the remote requires authentication to connect.
- `certificate_cb`: a pointer to a callback function which will be called if certificate verification fails. This lets the user decide whether or not to keep connecting. If the function returns 1, connecting will be allowed. If it returns 0, the connection will not be allowed. A negative value can be used to return errors.
- `payload`: the payload to be provided to the two callback functions.

Examples

```
julia> fo = LibGit2.FetchOptions(
           proxy_opts = LibGit2.ProxyOptions(url = Cstring("https://my_proxy_url.com")))

julia> fetch(remote, "master", options=fo)
```

`LibGit2.PushOptions` – Type.

```
LibGit2.PushOptions
```

Matches the `git_push_options` struct.

The fields represent:

- `version`: version of the struct in use, in case this changes later. For now, always 1.
- `parallelism`: if a pack file must be created, this variable sets the number of worker threads which will be spawned by the packbuilder. If 0, the packbuilder will auto-set the number of threads to use. The default is 1.
- `callbacks`: the callbacks (e.g. for authentication with the remote) to use for the push.

- `proxy_opts`: only relevant if the LibGit2 version is greater than or equal to `0.25.0`. Sets options for using a proxy to communicate with a remote. See `ProxyOptions` for more information.
- `custom_headers`: only relevant if the LibGit2 version is greater than or equal to `0.24.0`. Extra headers needed for the push operation.
- `remote_push_options`: only relevant if the LibGit2 version is greater than or equal to `1.8.0`. "Push options" to deliver to the remote.

`LibGit2.RebaseOperation` – Type.

```
LibGit2.RebaseOperation
```

Describes a single instruction/operation to be performed during the rebase. Matches the `git_rebase_operation` struct.

The fields represent:

- `optype`: the type of rebase operation currently being performed. The options are:
  - `REBASE_OPERATION_PICK`: cherry-pick the commit in question.
  - `REBASE_OPERATION_REWORD`: cherry-pick the commit in question, but rewrite its message using the prompt.
  - `REBASE_OPERATION_EDIT`: cherry-pick the commit in question, but allow the user to edit the commit's contents and its message.
  - `REBASE_OPERATION_SQUASH`: squash the commit in question into the previous commit. The commit messages of the two commits will be merged.
  - `REBASE_OPERATION_FIXUP`: squash the commit in question into the previous commit. Only the commit message of the previous commit will be used.
  - `REBASE_OPERATION_EXEC`: do not cherry-pick a commit. Run a command and continue if the command exits successfully.
- `id`: the `GitHash` of the commit being worked on during this rebase step.
- `exec`: in case `REBASE_OPERATION_EXEC` is used, the command to run during this step (for instance, running the test suite after each commit).

`LibGit2.RebaseOptions` – Type.

```
LibGit2.RebaseOptions
```

Matches the `git_rebase_options` struct.

The fields represent:

- `version`: version of the struct in use, in case this changes later. For now, always 1.
- `quiet`: inform other git clients helping with/working on the rebase that the rebase should be done "quietly". Used for interoperability. The default is 1.
- `inmemory`: start an in-memory rebase. Callers working on the rebase can go through its steps and commit any changes, but cannot rewind HEAD or update the repository. The `workdir` will not be modified. Only present on libgit2 versions newer than or equal to 0.24.0.
- `rewrite_notes_ref`: name of the reference to notes to use to rewrite the commit notes as the rebase is finished.

- `merge_opts`: merge options controlling how the trees will be merged at each rebase step. Only present on libgit2 versions newer than or equal to 0.24.0.
- `checkout_opts`: checkout options for writing files when initializing the rebase, stepping through it, and aborting it. See `CheckoutOptions` for more information.

`LibGit2.RemoteCallbacks` – Type.

```
LibGit2.RemoteCallbacks
```

Callback settings. Matches the `git_remote_callbacks` struct.

`LibGit2.SignatureStruct` – Type.

```
LibGit2.SignatureStruct
```

An action signature (e.g. for committers, taggers, etc). Matches the `git_signature` struct.

The fields represent:

- `name`: The full name of the committer or author of the commit.
- `email`: The email at which the committer/author can be contacted.
- `when`: a `TimeStruct` indicating when the commit was authored/committed into the repository.

`LibGit2.StatusEntry` – Type.

```
LibGit2.StatusEntry
```

Providing the differences between the file as it exists in HEAD and the index, and providing the differences between the index and the working directory. Matches the `git_status_entry` struct.

The fields represent:

- `status`: contains the status flags for the file, indicating if it is current, or has been changed in some way in the index or work tree.
- `head_to_index`: a pointer to a `DiffDelta` which encapsulates the difference(s) between the file as it exists in HEAD and in the index.
- `index_to_workdir`: a pointer to a `DiffDelta` which encapsulates the difference(s) between the file as it exists in the index and in the `workdir`.

`LibGit2.StatusOptions` – Type.

```
LibGit2.StatusOptions
```

Options to control how `git_status_foreach_ext()` will issue callbacks. Matches the `git_status_opt_t` struct.

The fields represent:

- `version`: version of the struct in use, in case this changes later. For now, always 1.
- `show`: a flag for which files to examine and in which order. The default is `Consts.STATUS_SHOW_INDEX_AND_WORKDIR`.
- `flags`: flags for controlling any callbacks used in a status call.

- pathspec: an array of paths to use for path-matching. The behavior of the path-matching will vary depending on the values of show and flags.
- The baseline is the tree to be used for comparison to the working directory and index; defaults to HEAD.

LibGit2.StrArrayStruct – Type.

```
LibGit2.StrArrayStruct
```

A LibGit2 representation of an array of strings. Matches the git_strarray struct.

When fetching data from LibGit2, a typical usage would look like:

```
sa_ref = Ref(StrArrayStruct())
@check ccall(..., (Ptr{StrArrayStruct},), sa_ref)
res = collect(sa_ref[])
free(sa_ref)
```

In particular, note that LibGit2.free should be called afterward on the Ref object.

Conversely, when passing a vector of strings to LibGit2, it is generally simplest to rely on implicit conversion:

```
strs = String[...]
@check ccall(..., (Ptr{StrArrayStruct},), strs)
```

Note that no call to free is required as the data is allocated by Julia.

LibGit2.TimeStruct – Type.

```
LibGit2.TimeStruct
```

Time in a signature. Matches the git_time struct.

LibGit2.addfile – Function.

```
addfile(cfg::GitConfig, path::AbstractString,
        level::Consts.GIT_CONFIG=Consts.CONFIG_LEVEL_APP,
        repo::Union{GitRepo, Nothing} = nothing,
        force::Bool=false)
```

Add an existing git configuration file located at path to the current GitConfig cfg. If the file does not exist, it will be created.

- level sets the git configuration priority level and is determined by

Consts.GIT_CONFIG.

- repo is an optional repository to allow parsing of conditional includes.
- If force is false and a configuration for the given priority level already exists,

addfile will error. If force is true, the existing configuration will be replaced by the one in the file at path.

LibGit2.add! – Function.

```
add!(repo::GitRepo, files::AbstractString...; flags::Cuint = Consts.INDEX_ADD_DEFAULT)
add!(idx::GitIndex, files::AbstractString...; flags::Cuint = Consts.INDEX_ADD_DEFAULT)
```

Add all the files with paths specified by `files` to the index `idx` (or the index of the `repo`). If the file already exists, the index entry will be updated. If the file does not exist already, it will be newly added into the index. `files` may contain glob patterns which will be expanded and any matching files will be added (unless `INDEX_ADD_DISABLE_PATHSPEC_MATCH` is set, see below). If a file has been ignored (in `.gitignore` or in the config), it *will not* be added, *unless* it is already being tracked in the index, in which case it *will* be updated. The keyword argument `flags` is a set of bit-flags which control the behavior with respect to ignored files:

- `Consts.INDEX_ADD_DEFAULT` - default, described above.
- `Consts.INDEX_ADD_FORCE` - disregard the existing ignore rules and force addition of the file to the index even if it is already ignored.
- `Consts.INDEX_ADD_CHECK_PATHSPEC` - cannot be used at the same time as `INDEX_ADD_FORCE`. Check that each file in `files` which exists on disk is not in the ignore list. If one of the files *is* ignored, the function will return `EINVALIDSPEC`.
- `Consts.INDEX_ADD_DISABLE_PATHSPEC_MATCH` - turn off glob matching, and only add files to the index which exactly match the paths specified in `files`.

LibGit2.add_fetch! – Function.

```
add_fetch!(repo::GitRepo, rmt::GitRemote, fetch_spec::String)
```

Add a *fetch* refspec for the specified `rmt`. This refspec will contain information about which branch(es) to fetch from.

Examples

```
julia> LibGit2.add_fetch!(repo, remote, "upstream");

julia> LibGit2.fetch_refspecs(remote)
String["+refs/heads/*:refs/remotes/upstream/*"]
```

LibGit2.add_push! – Function.

```
add_push!(repo::GitRepo, rmt::GitRemote, push_spec::String)
```

Add a *push* refspec for the specified `rmt`. This refspec will contain information about which branch(es) to push to.

Examples

```
julia> LibGit2.add_push!(repo, remote, "refs/heads/master");

julia> remote = LibGit2.get(LibGit2.GitRemote, repo, branch);

julia> LibGit2.push_refspecs(remote)
String["refs/heads/master"]
```

> **Note**
>
> You may need to `close` and reopen the `GitRemote` in question after updating its push refspecs in order for the change to take effect and for calls to `push` to work.

`LibGit2.addblob!` – Function.

```
LibGit2.addblob!(repo::GitRepo, path::AbstractString)
```

Read the file at `path` and adds it to the object database of `repo` as a loose blob. Return the `GitHash` of the resulting blob.

Examples

```
hash_str = string(commit_oid)
blob_file = joinpath(repo_path, ".git", "objects", hash_str[1:2], hash_str[3:end])
id = LibGit2.addblob!(repo, blob_file)
```

`LibGit2.author` – Function.

```
author(c::GitCommit)
```

Return the `Signature` of the author of the commit `c`. The author is the person who made changes to the relevant file(s). See also `committer`.

`LibGit2.authors` – Function.

```
authors(repo::GitRepo)::Vector{Signature}
```

Return all authors of commits to the `repo` repository.

Examples

```
repo = LibGit2.GitRepo(repo_path)
repo_file = open(joinpath(repo_path, test_file), "a")

println(repo_file, commit_msg)
flush(repo_file)
LibGit2.add!(repo, test_file)
sig = LibGit2.Signature("TEST", "TEST@TEST.COM", round(time(), 0), 0)
commit_oid1 = LibGit2.commit(repo, "commit1"; author=sig, committer=sig)
println(repo_file, randstring(10))
flush(repo_file)
LibGit2.add!(repo, test_file)
commit_oid2 = LibGit2.commit(repo, "commit2"; author=sig, committer=sig)

# will be a Vector of [sig, sig]
auths = LibGit2.authors(repo)
```

`LibGit2.branch` – Function.

```
branch(repo::GitRepo)
```

Equivalent to `git branch`. Create a new branch from the current HEAD.

`LibGit2.branch!` – Function.

```
branch!(repo::GitRepo, branch_name::AbstractString, commit::AbstractString=""; kwargs...)
```

Checkout a new git branch in the `repo` repository. `commit` is the `GitHash`, in string form, which will be the start of the new branch. If `commit` is an empty string, the current HEAD will be used.

The keyword arguments are:

- `track::AbstractString=""`: the name of the remote branch this new branch should track, if any. If empty (the default), no remote branch will be tracked.
- `force::Bool=false`: if `true`, branch creation will be forced.
- `set_head::Bool=true`: if `true`, after the branch creation finishes the branch head will be set as the HEAD of `repo`.

Equivalent to `git checkout [-b|-B] <branch_name> [<commit>] [--track <track>]`.

Examples

```
repo = LibGit2.GitRepo(repo_path)
LibGit2.branch!(repo, "new_branch", set_head=false)
```

`LibGit2.checkout!` – Function.

```
checkout!(repo::GitRepo, commit::AbstractString=""; force::Bool=true)
```

Equivalent to `git checkout [-f] --detach <commit>`. Checkout the git commit `commit` (a `GitHash` in string form) in `repo`. If `force` is `true`, force the checkout and discard any current changes. Note that this detaches the current HEAD.

Examples

```
repo = LibGit2.GitRepo(repo_path)
open(joinpath(LibGit2.path(repo), "file1"), "w") do f
    write(f, "111
")
end
LibGit2.add!(repo, "file1")
commit_oid = LibGit2.commit(repo, "add file1")
open(joinpath(LibGit2.path(repo), "file1"), "w") do f
    write(f, "112
")
end
# would fail without the force=true
# since there are modifications to the file
LibGit2.checkout!(repo, string(commit_oid), force=true)
```

`LibGit2.clone` – Function.

```
clone(repo_url::AbstractString, repo_path::AbstractString, clone_opts::CloneOptions)
```

Clone the remote repository at `repo_url` (which can be a remote URL or a path on the local filesystem) to `repo_path` (which must be a path on the local filesystem). Options for the clone, such as whether to perform a bare clone or not, are set by `CloneOptions`.

Examples

```
repo_url = "https://github.com/JuliaLang/Example.jl"
repo = LibGit2.clone(repo_url, "/home/me/projects/Example")
```

```
clone(repo_url::AbstractString, repo_path::AbstractString; kwargs...)
```

Clone a remote repository located at `repo_url` to the local filesystem location `repo_path`.

The keyword arguments are:

- `branch::AbstractString=""`: which branch of the remote to clone, if not the default repository branch (usually `master`).
- `isbare::Bool=false`: if `true`, clone the remote as a bare repository, which will make `repo_path` itself the git directory instead of `repo_path/.git`. This means that a working tree cannot be checked out. Plays the role of the git CLI argument `--bare`.
- `remote_cb::Ptr{Cvoid}=C_NULL`: a callback which will be used to create the remote before it is cloned. If `C_NULL` (the default), no attempt will be made to create the remote - it will be assumed to already exist.
- `depth::Integer=0`: create a shallow clone with a history truncated to the specified number of commits. `0` indicates a full clone (the default). Use `Consts.FETCH_DEPTH_UNSHALLOW` to fetch all missing data from a shallow clone. Note: shallow clones are, at the time of writing, only supported for network protocols (http, https, git, ssh), not for local filesystem paths. (https://github.com/libgit2/libgit2/issues/6634)
- `credentials::Creds=nothing`: provides credentials and/or settings when authenticating against a private repository.
- `callbacks::Callbacks=Callbacks()`: user provided callbacks and payloads.

Equivalent to `git clone [-b <branch>] [--bare] [--depth <depth>] <repo_url> <repo_path>`.

Examples

```
repo_url = "https://github.com/JuliaLang/Example.jl"
repo1 = LibGit2.clone(repo_url, "test_path")
repo2 = LibGit2.clone(repo_url, "test_path", isbare=true)
julia_url = "https://github.com/JuliaLang/julia"
julia_repo = LibGit2.clone(julia_url, "julia_path", branch="release-0.6")
# Shallow clone with only the most recent commit
shallow_repo = LibGit2.clone(repo_url, "shallow_path", depth=1)
```

`LibGit2.commit` – Function.

```
commit(repo::GitRepo, msg::AbstractString; kwargs...)::GitHash
```

Wrapper around `git_commit_create`. Create a commit in the repository `repo`. `msg` is the commit message. Return the OID of the new commit.

The keyword arguments are:

- `refname::AbstractString=Consts.HEAD_FILE`: if not NULL, the name of the reference to update to point to the new commit. For example, `"HEAD"` will update the HEAD of the current branch. If the reference does not yet exist, it will be created.
- `author::Signature = Signature(repo)` is a `Signature` containing information about the person who authored the commit.
- `committer::Signature = Signature(repo)` is a `Signature` containing information about the person who committed the commit to the repository. Not necessarily the same as `author`, for instance if `author` emailed a patch to `committer` who committed it.
- `tree_id::GitHash = GitHash()` is a git tree to use to create the commit, showing its ancestry and relationship with any other history. `tree` must belong to `repo`.
- `parent_ids::Vector{GitHash}=GitHash[]` is a list of commits by `GitHash` to use as parent commits for the new one, and may be empty. A commit might have multiple parents if it is a merge commit, for example.

```
LibGit2.commit(rb::GitRebase, sig::GitSignature)
```

Commit the current patch to the rebase `rb`, using `sig` as the committer. Is silent if the commit has already been applied.

`LibGit2.committer` – Function.

```
committer(c::GitCommit)
```

Return the `Signature` of the committer of the commit `c`. The committer is the person who committed the changes originally authored by the `author`, but need not be the same as the `author`, for example, if the `author` emailed a patch to a `committer` who committed it.

`LibGit2.count` – Function.

```
LibGit2.count(f::Function, walker::GitRevWalker; oid::GitHash=GitHash(),
↪  by::Cint=Consts.SORT_NONE, rev::Bool=false)
```

Using the `GitRevWalker` `walker` to "walk" over every commit in the repository's history, find the number of commits which return `true` when `f` is applied to them. The keyword arguments are: * `oid`: The `GitHash` of the commit to begin the walk from. The default is to use `push_head!` and therefore the HEAD commit and all its ancestors. * `by`: The sorting method. The default is not to sort. Other options are to sort by topology (`LibGit2.Consts.SORT_TOPOLOGICAL`), to sort forwards in time (`LibGit2.Consts.SORT_TIME`, most ancient first) or to sort backwards in time (`LibGit2.Consts.SORT_REVERSE`, most recent first). * `rev`: Whether to reverse the sorted order (for instance, if topological sorting is used).

Examples

```
cnt = LibGit2.with(LibGit2.GitRevWalker(repo)) do walker
    LibGit2.count((oid, repo)->(oid == commit_oid1), walker, oid=commit_oid1,
    ↪  by=LibGit2.Consts.SORT_TIME)
end
```

`LibGit2.count` finds the number of commits along the walk with a certain `GitHash` `commit_oid1`, starting the walk from that commit and moving forwards in time from it. Since the `GitHash` is unique to a commit, `cnt` will be 1.

`LibGit2.counthunks` – Function.

```
counthunks(blame::GitBlame)
```

Return the number of distinct "hunks" with a file. A hunk may contain multiple lines. A hunk is usually a piece of a file that was added/changed/removed together, for example, a function added to a source file or an inner loop that was optimized out of that function later.

`LibGit2.create_branch` – Function.

```
LibGit2.create_branch(repo::GitRepo, bname::AbstractString, commit_obj::GitCommit;
↪  force::Bool=false)
```

Create a new branch in the repository `repo` with name `bname`, which points to commit `commit_obj` (which has to be part of `repo`). If `force` is `true`, overwrite an existing branch named `bname` if it exists. If `force` is `false` and a branch already exists named `bname`, this function will throw an error.

`LibGit2.credentials_callback` – Function.

```
credential_callback(...)::Cint
```

A LibGit2 credential callback function which provides different credential acquisition functionality w.r.t. a connection protocol. The `payload_ptr` is required to contain a `LibGit2.CredentialPayload` object which will keep track of state and settings.

The `allowed_types` contains a bitmask of `LibGit2.Consts.GIT_CREDTYPE` values specifying which authentication methods should be attempted.

Credential authentication is done in the following order (if supported):

- SSH agent
- SSH private/public key pair
- Username/password plain text

If a user is presented with a credential prompt they can abort the prompt by typing `^D` (pressing the control key together with the `d` key).

Note: Due to the specifics of the `libgit2` authentication procedure, when authentication fails, this function is called again without any indication whether authentication was successful or not. To avoid an infinite loop from repeatedly using the same faulty credentials, we will keep track of state using the payload.

For addition details see the LibGit2 guide on authenticating against a server.

`LibGit2.credentials_cb` – Function.

C function pointer for `credentials_callback`

`LibGit2.default_signature` – Function.

Return signature object. Free it after use.

`LibGit2.delete_branch` – Function.

```
LibGit2.delete_branch(branch::GitReference)
```

Delete the branch pointed to by branch.

LibGit2.diff_files – Function.

```
diff_files(repo::GitRepo, branch1::AbstractString, branch2::AbstractString;
↪  kwarg...)::Vector{AbstractString}
```

Show which files have changed in the git repository repo between branches branch1 and branch2.

The keyword argument is:

- filter::Set{Consts.DELTA_STATUS}=Set([Consts.DELTA_ADDED, Consts.DELTA_MODIFIED, Consts.DELTA_DELET and it sets options for the diff. The default is to show files added, modified, or deleted.

Return only the *names* of the files which have changed, *not* their contents.

Examples

```
LibGit2.branch!(repo, "branch/a")
LibGit2.branch!(repo, "branch/b")
# add a file to repo
open(joinpath(LibGit2.path(repo),"file"),"w") do f
    write(f, "hello repo
")
end
LibGit2.add!(repo, "file")
LibGit2.commit(repo, "add file")
# returns ["file"]
filt = Set([LibGit2.Consts.DELTA_ADDED])
files = LibGit2.diff_files(repo, "branch/a", "branch/b", filter=filt)
# returns [] because existing files weren't modified
filt = Set([LibGit2.Consts.DELTA_MODIFIED])
files = LibGit2.diff_files(repo, "branch/a", "branch/b", filter=filt)
```

Equivalent to git diff --name-only --diff-filter=<filter> <branch1> <branch2>.

LibGit2.entryid – Function.

```
entryid(te::GitTreeEntry)
```

Return the GitHash of the object to which te refers.

LibGit2.entrytype – Function.

```
entrytype(te::GitTreeEntry)
```

Return the type of the object to which te refers. The result will be one of the types which objtype returns, e.g. a GitTree or GitBlob.

LibGit2.fetch – Function.

```
fetch(rmt::GitRemote, refspecs; options::FetchOptions=FetchOptions(), msg="")
```

Fetch from the specified rmt remote git repository, using refspecs to determine which remote branch(es) to fetch. The keyword arguments are:

- `options`: determines the options for the fetch, e.g. whether to prune afterwards. See `FetchOptions` for more information.
- `msg`: a message to insert into the reflogs.

```
fetch(repo::GitRepo; kwargs...)
```

Fetches updates from an upstream of the repository `repo`.

The keyword arguments are:

- `remote::AbstractString="origin"`: which remote, specified by name, of `repo` to fetch from. If this is empty, the URL will be used to construct an anonymous remote.
- `remoteurl::AbstractString=""`: the URL of `remote`. If not specified, will be assumed based on the given name of `remote`.
- `refspecs=AbstractString[]`: determines properties of the fetch.
- `depth::Integer=0`: limit fetching to the specified number of commits from the tip of each remote branch. `0` indicates a full fetch (the default). Use `Consts.FETCH_DEPTH_UNSHALLOW` to fetch all missing data from a shallow clone. Note: depth is, at the time of writing, only supported for network protocols (http, https, git, ssh), not for local filesystem paths. (https://github.com/libgit2/libgit2/issues/6634)
- `credentials=nothing`: provides credentials and/or settings when authenticating against a private `remote`.
- `callbacks=Callbacks()`: user provided callbacks and payloads.

Equivalent to `git fetch [--depth <depth>] [<remoteurl>|<repo>] [<refspecs>]`.

`LibGit2.fetchheads` – Function.

```
fetchheads(repo::GitRepo)::Vector{FetchHead}
```

Return the list of all the fetch heads for `repo`, each represented as a `FetchHead`, including their names, URLs, and merge statuses.

Examples

```
julia> fetch_heads = LibGit2.fetchheads(repo);

julia> fetch_heads[1].name
"refs/heads/master"

julia> fetch_heads[1].ismerge
true

julia> fetch_heads[2].name
"refs/heads/test_branch"

julia> fetch_heads[2].ismerge
false
```

`LibGit2.fetch_refspecs` – Function.

```
fetch_refspecs(rmt::GitRemote)::Vector{String}
```

Get the *fetch* refspecs for the specified `rmt`. These refspecs contain information about which branch(es) to fetch from.

Examples

```
julia> remote = LibGit2.get(LibGit2.GitRemote, repo, "upstream");

julia> LibGit2.add_fetch!(repo, remote, "upstream");

julia> LibGit2.fetch_refspecs(remote)
String["+refs/heads/*:refs/remotes/upstream/*"]
```

`LibGit2.fetchhead_foreach_cb` – Function.

C function pointer for `fetchhead_foreach_callback`

`LibGit2.merge_base` – Function.

```
merge_base(repo::GitRepo, one::AbstractString, two::AbstractString)::GitHash
```

Find a merge base (a common ancestor) between the commits `one` and `two`. `one` and `two` may both be in string form. Return the `GitHash` of the merge base.

`LibGit2.merge!` – Method.

```
merge!(repo::GitRepo; kwargs...)::Bool
```

Perform a git merge on the repository `repo`, merging commits with diverging history into the current branch. Return `true` if the merge succeeded, `false` if not.

The keyword arguments are:

- `committish::AbstractString=""`: Merge the named commit(s) in `committish`.
- `branch::AbstractString=""`: Merge the branch `branch` and all its commits since it diverged from the current branch.
- `fastforward::Bool=false`: If `fastforward` is `true`, only merge if the merge is a fast-forward (the current branch head is an ancestor of the commits to be merged), otherwise refuse to merge and return `false`. This is equivalent to the git CLI option `--ff-only`.
- `merge_opts::MergeOptions=MergeOptions()`: `merge_opts` specifies options for the merge, such as merge strategy in case of conflicts.
- `checkout_opts::CheckoutOptions=CheckoutOptions()`: `checkout_opts` specifies options for the checkout step.

Equivalent to `git merge [--ff-only] [<committish> | <branch>]`.

Note

If you specify a `branch`, this must be done in reference format, since the string will be turned into a `GitReference`. For example, if you wanted to merge branch `branch_a`, you would call `merge!(repo, branch="refs/heads/branch_a")`.

LibGit2.merge! – Method.

```
merge!(repo::GitRepo, anns::Vector{GitAnnotated}; kwargs...)::Bool
```

Merge changes from the annotated commits (captured as GitAnnotated objects) anns into the HEAD of the repository repo. The keyword arguments are:

- merge_opts::MergeOptions = MergeOptions(): options for how to perform the merge, including whether fastforwarding is allowed. See MergeOptions for more information.
- checkout_opts::CheckoutOptions = CheckoutOptions(): options for how to perform the checkout. See CheckoutOptions for more information.

anns may refer to remote or local branch heads. Return true if the merge is successful, otherwise return false (for instance, if no merge is possible because the branches have no common ancestor).

Examples

```
upst_ann = LibGit2.GitAnnotated(repo, "branch/a")

# merge the branch in
LibGit2.merge!(repo, [upst_ann])
```

LibGit2.merge! – Method.

```
merge!(repo::GitRepo, anns::Vector{GitAnnotated}, fastforward::Bool; kwargs...)::Bool
```

Merge changes from the annotated commits (captured as GitAnnotated objects) anns into the HEAD of the repository repo. If fastforward is true, *only* a fastforward merge is allowed. In this case, if conflicts occur, the merge will fail. Otherwise, if fastforward is false, the merge may produce a conflict file which the user will need to resolve.

The keyword arguments are:

- merge_opts::MergeOptions = MergeOptions(): options for how to perform the merge, including whether fastforwarding is allowed. See MergeOptions for more information.
- checkout_opts::CheckoutOptions = CheckoutOptions(): options for how to perform the checkout. See CheckoutOptions for more information.

anns may refer to remote or local branch heads. Return true if the merge is successful, otherwise return false (for instance, if no merge is possible because the branches have no common ancestor).

Examples

```
upst_ann_1 = LibGit2.GitAnnotated(repo, "branch/a")

# merge the branch in, fastforward
LibGit2.merge!(repo, [upst_ann_1], true)

# merge conflicts!
upst_ann_2 = LibGit2.GitAnnotated(repo, "branch/b")
# merge the branch in, try to fastforward
LibGit2.merge!(repo, [upst_ann_2], true) # will return false
LibGit2.merge!(repo, [upst_ann_2], false) # will return true
```

`LibGit2.ffmerge!` – Function.

```
ffmerge!(repo::GitRepo, ann::GitAnnotated)
```

Fastforward merge changes into current HEAD. This is only possible if the commit referred to by `ann` is descended from the current HEAD (e.g. if pulling changes from a remote branch which is simply ahead of the local branch tip).

`LibGit2.fullname` – Function.

```
LibGit2.fullname(ref::GitReference)
```

Return the name of the reference pointed to by the symbolic reference `ref`. If `ref` is not a symbolic reference, return an empty string.

`LibGit2.features` – Function.

```
features()
```

Return a list of git features the current version of libgit2 supports, such as threading or using HTTPS or SSH.

`LibGit2.filename` – Function.

```
filename(te::GitTreeEntry)
```

Return the filename of the object on disk to which `te` refers.

`LibGit2.filemode` – Function.

```
filemode(te::GitTreeEntry)::Cint
```

Return the UNIX filemode of the object on disk to which `te` refers as an integer.

`LibGit2.gitdir` – Function.

```
LibGit2.gitdir(repo::GitRepo)
```

Return the location of the "git" files of `repo`:

- for normal repositories, this is the location of the `.git` folder.
- for bare repositories, this is the location of the repository itself.

See also `workdir`, `path`.

`LibGit2.git_url` – Function.

```
LibGit2.git_url(; kwargs...)::String
```

Create a string based upon the URL components provided. When the `scheme` keyword is not provided the URL produced will use the alternative scp-like syntax.

Keywords

- `scheme::AbstractString=""`: the URL scheme which identifies the protocol to be used. For HTTP use "http", SSH use "ssh", etc. When `scheme` is not provided the output format will be "ssh" but using the scp-like syntax.
- `username::AbstractString=""`: the username to use in the output if provided.
- `password::AbstractString=""`: the password to use in the output if provided.
- `host::AbstractString=""`: the hostname to use in the output. A hostname is required to be specified.
- `port::Union{AbstractString,Integer}=""`: the port number to use in the output if provided. Cannot be specified when using the scp-like syntax.
- `path::AbstractString=""`: the path to use in the output if provided.

Warning

Avoid using passwords in URLs. Unlike the credential objects, Julia is not able to securely zero or destroy the sensitive data after use and the password may remain in memory; possibly to be exposed by an uninitialized memory.

Examples

```
julia> LibGit2.git_url(username="git", host="github.com", path="JuliaLang/julia.git")
"git@github.com:JuliaLang/julia.git"

julia> LibGit2.git_url(scheme="https", host="github.com", path="/JuliaLang/julia.git")
"https://github.com/JuliaLang/julia.git"

julia> LibGit2.git_url(scheme="ssh", username="git", host="github.com", port=2222,
↪  path="JuliaLang/julia.git")
"ssh://git@github.com:2222/JuliaLang/julia.git"
```

`LibGit2.@githash_str` – Macro.

```
@githash_str -> AbstractGitHash
```

Construct a git hash object from the given string, returning a `GitShortHash` if the string is shorter than 40 hexadecimal digits, otherwise a `GitHash`.

Examples

```
julia> LibGit2.githash"d114feb74ce633"
GitShortHash("d114feb74ce633")

julia> LibGit2.githash"d114feb74ce63307afe878a5228ad014e0289a85"
GitHash("d114feb74ce63307afe878a5228ad014e0289a85")
```

`LibGit2.head` – Function.

```
LibGit2.head(repo::GitRepo)::GitReference
```

Return a `GitReference` to the current HEAD of `repo`.

```
head(pkg::AbstractString)::String
```

Return current HEAD GitHash of the pkg repo as a string.

LibGit2.head! – Function.

```
LibGit2.head!(repo::GitRepo, ref::GitReference)::GitReference
```

Set the HEAD of repo to the object pointed to by ref.

LibGit2.head_oid – Function.

```
LibGit2.head_oid(repo::GitRepo)::GitHash
```

Lookup the object id of the current HEAD of git repository repo.

LibGit2.headname – Function.

```
LibGit2.headname(repo::GitRepo)
```

Lookup the name of the current HEAD of git repository repo. If repo is currently detached, return the name of the HEAD it's detached from.

LibGit2.init – Function.

```
LibGit2.init(path::AbstractString, bare::Bool=false)::GitRepo
```

Open a new git repository at path. If bare is false, the working tree will be created in path/.git. If bare is true, no working directory will be created.

LibGit2.is_ancestor_of – Function.

```
is_ancestor_of(a::AbstractString, b::AbstractString, repo::GitRepo)::Bool
```

Return true if a, a GitHash in string form, is an ancestor of b, a GitHash in string form.

Examples

```
julia> repo = GitRepo(repo_path);

julia> LibGit2.add!(repo, test_file1);

julia> commit_oid1 = LibGit2.commit(repo, "commit1");

julia> LibGit2.add!(repo, test_file2);

julia> commit_oid2 = LibGit2.commit(repo, "commit2");

julia> LibGit2.is_ancestor_of(string(commit_oid1), string(commit_oid2), repo)
true
```

LibGit2.isbinary – Function.

```
isbinary(blob::GitBlob)::Bool
```

Use a heuristic to guess if a file is binary: searching for NULL bytes and looking for a reasonable ratio of printable to non-printable characters among the first 8000 bytes.

`LibGit2.iscommit` – Function.

```
iscommit(id::AbstractString, repo::GitRepo)::Bool
```

Check if commit `id` (which is a `GitHash` in string form) is in the repository.

Examples

```
julia> repo = GitRepo(repo_path);

julia> LibGit2.add!(repo, test_file);

julia> commit_oid = LibGit2.commit(repo, "add test_file");

julia> LibGit2.iscommit(string(commit_oid), repo)
true
```

`LibGit2.isdiff` – Function.

```
LibGit2.isdiff(repo::GitRepo, treeish::AbstractString, pathspecs::AbstractString="";
↪  cached::Bool=false)
```

Checks if there are any differences between the tree specified by `treeish` and the tracked files in the working tree (if `cached=false`) or the index (if `cached=true`). `pathspecs` are the specifications for options for the diff.

Examples

```
repo = LibGit2.GitRepo(repo_path)
LibGit2.isdiff(repo, "HEAD") # should be false
open(joinpath(repo_path, new_file), "a") do f
    println(f, "here's my cool new file")
end
LibGit2.isdiff(repo, "HEAD") # now true
```

Equivalent to `git diff-index <treeish> [-- <pathspecs>]`.

`LibGit2.isdirty` – Function.

```
LibGit2.isdirty(repo::GitRepo, pathspecs::AbstractString=""; cached::Bool=false)::Bool
```

Check if there have been any changes to tracked files in the working tree (if `cached=false`) or the index (if `cached=true`). `pathspecs` are the specifications for options for the diff.

Examples

```
repo = LibGit2.GitRepo(repo_path)
LibGit2.isdirty(repo) # should be false
open(joinpath(repo_path, new_file), "a") do f
    println(f, "here's my cool new file")
end
LibGit2.isdirty(repo) # now true
LibGit2.isdirty(repo, new_file) # now true
```

Equivalent to `git diff-index HEAD [-- <pathspecs>]`.

LibGit2.isorphan – Function.

```
LibGit2.isorphan(repo::GitRepo)
```

Check if the current branch is an "orphan" branch, i.e. has no commits. The first commit to this branch will have no parents.

LibGit2.isset – Function.

```
isset(val::Integer, flag::Integer)
```

Test whether the bits of `val` indexed by `flag` are set (`1`) or unset (`0`).

LibGit2.iszero – Function.

```
iszero(id::GitHash)::Bool
```

Determine whether all hexadecimal digits of the given `GitHash` are zero.

LibGit2.lookup_branch – Function.

```
lookup_branch(repo::GitRepo, branch_name::AbstractString,
↪  remote::Bool=false)::Union{GitReference, Nothing}
```

Determine if the branch specified by `branch_name` exists in the repository `repo`. If `remote` is `true`, `repo` is assumed to be a remote git repository. Otherwise, it is part of the local filesystem.

Return either a `GitReference` to the requested branch if it exists, or `nothing` if not.

LibGit2.map – Function.

```
LibGit2.map(f::Function, walker::GitRevWalker; oid::GitHash=GitHash(),
↪  range::AbstractString="", by::Cint=Consts.SORT_NONE, rev::Bool=false)
```

Using the `GitRevWalker` `walker` to "walk" over every commit in the repository's history, apply `f` to each commit in the walk. The keyword arguments are: * `oid`: The `GitHash` of the commit to begin the walk from. The default is to use `push_head!` and therefore the HEAD commit and all its ancestors. * `range`: A range of `GitHashs` in the format `oid1..oid2`. `f` will be applied to all commits between the two. * `by`: The sorting method. The default is not to sort. Other options are to sort by topology (`LibGit2.Consts.SORT_TOPOLOGICAL`), to sort forwards in time (`LibGit2.Consts.SORT_TIME`, most ancient first) or to sort backwards in time (`LibGit2.Consts.SORT_REVERSE`, most recent first). * `rev`: Whether to reverse the sorted order (for instance, if topological sorting is used).

Examples

```
oids = LibGit2.with(LibGit2.GitRevWalker(repo)) do walker
    LibGit2.map((oid, repo)->string(oid), walker, by=LibGit2.Consts.SORT_TIME)
end
```

Here, `LibGit2.map` visits each commit using the `GitRevWalker` and finds its `GitHash`.

LibGit2.mirror_callback – Function.

Mirror callback function

Function sets `+refs/*:refs/*` refspecs and `mirror` flag for remote reference.

`LibGit2.mirror_cb` – Function.

C function pointer for `mirror_callback`

`LibGit2.message` – Function.

```
message(c::GitCommit, raw::Bool=false)
```

Return the commit message describing the changes made in commit `c`. If `raw` is `false`, return a slightly "cleaned up" message (which has any leading newlines removed). If `raw` is `true`, the message is not stripped of any such newlines.

`LibGit2.merge_analysis` – Function.

```
merge_analysis(repo::GitRepo, anns::Vector{GitAnnotated}) -> analysis, preference
```

Run analysis on the branches pointed to by the annotated branch tips anns and determine under what circumstances they can be merged. For instance, if `anns[1]` is simply an ancestor of `ann[2]`, then `merge_analysis` will report that a fast-forward merge is possible.

Return two outputs, `analysis` and `preference`. `analysis` has several possible values: * `MERGE_ANALYSIS_NONE`: it is not possible to merge the elements of `anns`. * `MERGE_ANALYSIS_NORMAL`: a regular merge, when HEAD and the commits that the user wishes to merge have all diverged from a common ancestor. In this case the changes have to be resolved and conflicts may occur. * `MERGE_ANALYSIS_UP_TO_DATE`: all the input commits the user wishes to merge can be reached from HEAD, so no merge needs to be performed. * `MERGE_ANALYSIS_FASTFORWARD`: the input commit is a descendant of HEAD and so no merge needs to be performed - instead, the user can simply checkout the input commit(s). * `MERGE_ANALYSIS_UNBORN`: the HEAD of the repository refers to a commit which does not exist. It is not possible to merge, but it may be possible to checkout the input commits. `preference` also has several possible values: * `MERGE_PREFERENCE_NONE`: the user has no preference. * `MERGE_PREFERENCE_NO_FASTFORWARD`: do not allow any fast-forward merges. * `MERGE_PREFERENCE_FASTFORWARD_ONLY`: allow only fast-forward merges and no other type (which may introduce conflicts). `preference` can be controlled through the repository or global git configuration.

`LibGit2.name` – Function.

```
LibGit2.name(ref::GitReference)
```

Return the full name of `ref`.

```
name(rmt::GitRemote)
```

Get the name of a remote repository, for instance `"origin"`. If the remote is anonymous (see `GitRemoteAnon`) the name will be an empty string `""`.

Examples

```
julia> repo_url = "https://github.com/JuliaLang/Example.jl";

julia> repo = LibGit2.clone(cache_repo, "test_directory");

julia> remote = LibGit2.GitRemote(repo, "origin", repo_url);

julia> name(remote)
"origin"
```

```
LibGit2.name(tag::GitTag)
```

The name of `tag` (e.g. `"v0.5"`).

`LibGit2.need_update` – Function.

```
need_update(repo::GitRepo)
```

Equivalent to `git update-index`. Return `true` if `repo` needs updating.

`LibGit2.objtype` – Function.

```
objtype(obj_type::Consts.OBJECT)
```

Return the type corresponding to the enum value.

`LibGit2.path` – Function.

```
LibGit2.path(repo::GitRepo)
```

Return the base file path of the repository `repo`.

- for normal repositories, this will typically be the parent directory of the ".git" directory (note: this may be different than the working directory, see `workdir` for more details).
- for bare repositories, this is the location of the "git" files.

See also `gitdir`, `workdir`.

`LibGit2.peel` – Function.

```
peel([T,] ref::GitReference)
```

Recursively peel `ref` until an object of type `T` is obtained. If no `T` is provided, then `ref` will be peeled until an object other than a `GitTag` is obtained.

- A `GitTag` will be peeled to the object it references.
- A `GitCommit` will be peeled to a `GitTree`.

Note

Only annotated tags can be peeled to `GitTag` objects. Lightweight tags (the default) are references under `refs/tags/` which point directly to `GitCommit` objects.

```
peel([T,] obj::GitObject)
```

Recursively peel `obj` until an object of type `T` is obtained. If no `T` is provided, then `obj` will be peeled until the type changes.

- A `GitTag` will be peeled to the object it references.
- A `GitCommit` will be peeled to a `GitTree`.

LibGit2.posixpath – Function.

```
LibGit2.posixpath(path)
```

Standardise the path string path to use POSIX separators.

LibGit2.push – Function.

```
push(rmt::GitRemote, refspecs; force::Bool=false, options::PushOptions=PushOptions())
```

Push to the specified rmt remote git repository, using refspecs to determine which remote branch(es) to push to. The keyword arguments are:

- force: if true, a force-push will occur, disregarding conflicts.
- options: determines the options for the push, e.g. which proxy headers to use. See PushOptions for more information.

Note

You can add information about the push refspecs in two other ways: by setting an option in the repository's GitConfig (with push.default as the key) or by calling add_push!. Otherwise you will need to explicitly specify a push refspec in the call to push for it to have any effect, like so: LibGit2.push(repo, refspecs=["refs/heads/master"]).

```
push(repo::GitRepo; kwargs...)
```

Pushes updates to an upstream of repo.

The keyword arguments are:

- remote::AbstractString="origin": the name of the upstream remote to push to.
- remoteurl::AbstractString="": the URL of remote.
- refspecs=AbstractString[]: determines properties of the push.
- force::Bool=false: determines if the push will be a force push, overwriting the remote branch.
- credentials=nothing: provides credentials and/or settings when authenticating against a private remote.
- callbacks=Callbacks(): user provided callbacks and payloads.

Equivalent to git push [<remoteurl>|<repo>] [<refspecs>].

LibGit2.push! – Method.

```
LibGit2.push!(w::GitRevWalker, cid::GitHash)
```

Start the GitRevWalker walker at commit cid. This function can be used to apply a function to all commits since a certain year, by passing the first commit of that year as cid and then passing the resulting w to LibGit2.map.

LibGit2.push_head! – Function.

```
LibGit2.push_head!(w::GitRevWalker)
```

Push the HEAD commit and its ancestors onto the `GitRevWalker` `w`. This ensures that HEAD and all its ancestor commits will be encountered during the walk.

`LibGit2.push_refspecs` – Function.

```
push_refspecs(rmt::GitRemote)::Vector{String}
```

Get the *push* refspecs for the specified `rmt`. These refspecs contain information about which branch(es) to push to.

Examples

```
julia> remote = LibGit2.get(LibGit2.GitRemote, repo, "upstream");

julia> LibGit2.add_push!(repo, remote, "refs/heads/master");

julia> close(remote);

julia> remote = LibGit2.get(LibGit2.GitRemote, repo, "upstream");

julia> LibGit2.push_refspecs(remote)
String["refs/heads/master"]
```

`LibGit2.raw` – Function.

```
raw(id::GitHash)::Vector{UInt8}
```

Obtain the raw bytes of the `GitHash` as a vector of length 20.

`LibGit2.read_tree!` – Function.

```
LibGit2.read_tree!(idx::GitIndex, tree::GitTree)
LibGit2.read_tree!(idx::GitIndex, treehash::AbstractGitHash)
```

Read the tree `tree` (or the tree pointed to by `treehash` in the repository owned by `idx`) into the index `idx`. The current index contents will be replaced.

`LibGit2.rebase!` – Function.

```
LibGit2.rebase!(repo::GitRepo, upstream::AbstractString="", newbase::AbstractString="")
```

Attempt an automatic merge rebase of the current branch, from `upstream` if provided, or otherwise from the upstream tracking branch. `newbase` is the branch to rebase onto. By default this is `upstream`.

If any conflicts arise which cannot be automatically resolved, the rebase will abort, leaving the repository and working tree in its original state, and the function will throw a `GitError`. This is roughly equivalent to the following command line statement:

```
git rebase --merge [<upstream>]
if [ -d ".git/rebase-merge" ]; then
    git rebase --abort
fi
```

`LibGit2.ref_list` – Function.

```
LibGit2.ref_list(repo::GitRepo)::Vector{String}
```

Get a list of all reference names in the `repo` repository.

`LibGit2.reftype` – Function.

```
LibGit2.reftype(ref::GitReference)::Cint
```

Return a `Cint` corresponding to the type of `ref`:

- `0` if the reference is invalid
- `1` if the reference is an object id
- `2` if the reference is symbolic

`LibGit2.remotes` – Function.

```
LibGit2.remotes(repo::GitRepo)
```

Return a vector of the names of the remotes of `repo`.

`LibGit2.remove!` – Function.

```
remove!(repo::GitRepo, files::AbstractString...)
remove!(idx::GitIndex, files::AbstractString...)
```

Remove all the files with paths specified by `files` in the index `idx` (or the index of the `repo`).

`LibGit2.reset` – Function.

```
reset(val::Integer, flag::Integer)
```

Unset the bits of `val` indexed by `flag`, returning them to `0`.

`LibGit2.reset!` – Function.

```
reset!(payload, [config])::CredentialPayload
```

Reset the `payload` state back to the initial values so that it can be used again within the credential callback. If a `config` is provided the configuration will also be updated.

Updates some entries, determined by the `pathspecs`, in the index from the target commit tree.

Sets the current head to the specified commit oid and optionally resets the index and working tree to match.

git reset [<committish>] [–] <pathspecs>...

```
reset!(repo::GitRepo, id::GitHash, mode::Cint=Consts.RESET_MIXED)
```

Reset the repository `repo` to its state at `id`, using one of three modes set by `mode`:

1. `Consts.RESET_SOFT` - move HEAD to `id`.

2. `Consts.RESET_MIXED` - default, move HEAD to `id` and reset the index to `id`.
3. `Consts.RESET_HARD` - move HEAD to `id`, reset the index to `id`, and discard all working changes.

Examples

```
# fetch changes
LibGit2.fetch(repo)
isfile(joinpath(repo_path, our_file)) # will be false

# fastforward merge the changes
LibGit2.merge!(repo, fastforward=true)

# because there was not any file locally, but there is
# a file remotely, we need to reset the branch
head_oid = LibGit2.head_oid(repo)
new_head = LibGit2.reset!(repo, head_oid, LibGit2.Consts.RESET_HARD)
```

In this example, the remote which is being fetched from *does* have a file called `our_file` in its index, which is why we must reset.

Equivalent to `git reset [--soft | --mixed | --hard] <id>`.

Examples

```
repo = LibGit2.GitRepo(repo_path)
head_oid = LibGit2.head_oid(repo)
open(joinpath(repo_path, "file1"), "w") do f
    write(f, "111
")
end
LibGit2.add!(repo, "file1")
mode = LibGit2.Consts.RESET_HARD
# will discard the changes to file1
# and unstage it
new_head = LibGit2.reset!(repo, head_oid, mode)
```

`LibGit2.restore` – Function.

```
restore(s::State, repo::GitRepo)
```

Return a repository `repo` to a previous `State` `s`, for example the HEAD of a branch before a merge attempt. `s` can be generated using the `snapshot` function.

`LibGit2.revcount` – Function.

```
LibGit2.revcount(repo::GitRepo, commit1::AbstractString, commit2::AbstractString)
```

List the number of revisions between `commit1` and `commit2` (committish OIDs in string form). Since `commit1` and `commit2` may be on different branches, `revcount` performs a "left-right" revision list (and count), returning a tuple of `Int`s - the number of left and right commits, respectively. A left (or right) commit refers to which side of a symmetric difference in a tree the commit is reachable from.

Equivalent to `git rev-list --left-right --count <commit1> <commit2>`.

Examples

```
repo = LibGit2.GitRepo(repo_path)
repo_file = open(joinpath(repo_path, test_file), "a")
println(repo_file, "hello world")
flush(repo_file)
LibGit2.add!(repo, test_file)
commit_oid1 = LibGit2.commit(repo, "commit 1")
println(repo_file, "hello world again")
flush(repo_file)
LibGit2.add!(repo, test_file)
commit_oid2 = LibGit2.commit(repo, "commit 2")
LibGit2.revcount(repo, string(commit_oid1), string(commit_oid2))
```

This will return `(-1, 0)`.

`LibGit2.set_remote_url` – Function.

```
set_remote_url(repo::GitRepo, remote_name, url)
set_remote_url(repo::String, remote_name, url)
```

Set both the fetch and push `url` for `remote_name` for the `GitRepo` or the git repository located at `path`. Typically git repos use `"origin"` as the remote name.

Examples

```
repo_path = joinpath(tempdir(), "Example")
repo = LibGit2.init(repo_path)
LibGit2.set_remote_url(repo, "upstream", "https://github.com/JuliaLang/Example.jl")
LibGit2.set_remote_url(repo_path, "upstream2", "https://github.com/JuliaLang/Example2.jl")
```

`LibGit2.shortname` – Function.

```
LibGit2.shortname(ref::GitReference)
```

Return a shortened version of the name of `ref` that's "human-readable".

```
julia> repo = GitRepo(path_to_repo);

julia> branch_ref = LibGit2.head(repo);

julia> LibGit2.name(branch_ref)
"refs/heads/master"

julia> LibGit2.shortname(branch_ref)
"master"
```

`LibGit2.snapshot` – Function.

```
snapshot(repo::GitRepo)::State
```

Take a snapshot of the current state of the repository `repo`, storing the current HEAD, index, and any uncommitted work. The output `State` can be used later during a call to `restore` to return the repository to the snapshotted state.

`LibGit2.split_cfg_entry` – Function.

```
LibGit2.split_cfg_entry(ce::LibGit2.ConfigEntry)::Tuple{String,String,String,String}
```

Break the `ConfigEntry` up to the following pieces: section, subsection, name, and value.

Examples

Given the git configuration file containing:

```
[credential "https://example.com"]
    username = me
```

The `ConfigEntry` would look like the following:

```
julia> entry
ConfigEntry("credential.https://example.com.username", "me")

julia> LibGit2.split_cfg_entry(entry)
("credential", "https://example.com", "username", "me")
```

Refer to the git config syntax documentation for more details.

`LibGit2.status` – Function.

```
LibGit2.status(repo::GitRepo, path::String)::Union{Cuint, Cvoid}
```

Lookup the status of the file at `path` in the git repository `repo`. For instance, this can be used to check if the file at `path` has been modified and needs to be staged and committed.

`LibGit2.stage` – Function.

```
stage(ie::IndexEntry)::Cint
```

Get the stage number of `ie`. The stage number `0` represents the current state of the working tree, but other numbers can be used in the case of a merge conflict. In such a case, the various stage numbers on an `IndexEntry` describe which side(s) of the conflict the current state of the file belongs to. Stage `0` is the state before the attempted merge, stage 1 is the changes which have been made locally, stages 2 and larger are for changes from other branches (for instance, in the case of a multi-branch "octopus" merge, stages 2, 3, and 4 might be used).

`LibGit2.tag_create` – Function.

```
LibGit2.tag_create(repo::GitRepo, tag::AbstractString, commit; kwargs...)
```

Create a new git tag `tag` (e.g. `"v0.5"`) in the repository `repo`, at the commit `commit`.

The keyword arguments are:

- `msg::AbstractString=""`: the message for the tag.
- `force::Bool=false`: if `true`, existing references will be overwritten.
- `sig::Signature=Signature(repo)`: the tagger's signature.

`LibGit2.tag_delete` – Function.

```
LibGit2.tag_delete(repo::GitRepo, tag::AbstractString)
```

Remove the git tag `tag` from the repository `repo`.

`LibGit2.tag_list` – Function.

```
LibGit2.tag_list(repo::GitRepo)::Vector{String}
```

Get a list of all tags in the git repository `repo`.

`LibGit2.target` – Function.

```
LibGit2.target(tag::GitTag)
```

The `GitHash` of the target object of `tag`.

`LibGit2.toggle` – Function.

```
toggle(val::Integer, flag::Integer)
```

Flip the bits of `val` indexed by `flag`, so that if a bit is `0` it will be `1` after the toggle, and vice-versa.

`LibGit2.transact` – Function.

```
transact(f::Function, repo::GitRepo)
```

Apply function `f` to the git repository `repo`, taking a `snapshot` before applying `f`. If an error occurs within `f`, `repo` will be returned to its snapshot state using `restore`. The error which occurred will be rethrown, but the state of `repo` will not be corrupted.

`LibGit2.treewalk` – Function.

```
treewalk(f, tree::GitTree, post::Bool=false)
```

Traverse the entries in `tree` and its subtrees in post or pre order. Preorder means beginning at the root and then traversing the leftmost subtree (and recursively on down through that subtree's leftmost subtrees) and moving right through the subtrees. Postorder means beginning at the bottom of the leftmost subtree, traversing upwards through it, then traversing the next right subtree (again beginning at the bottom) and finally visiting the tree root last of all.

The function parameter `f` should have following signature:

```
(String, GitTreeEntry) -> Cint
```

A negative value returned from `f` stops the tree walk. A positive value means that the entry will be skipped if `post` is `false`.

`LibGit2.upstream` – Function.

```
upstream(ref::GitReference)::Union{GitReference, Nothing}
```

Determine if the branch containing `ref` has a specified upstream branch.

Return either a `GitReference` to the upstream branch if it exists, or `nothing` if the requested branch does not have an upstream counterpart.

`LibGit2.update!` – Function.

```
update!(repo::GitRepo, files::AbstractString...)
update!(idx::GitIndex, files::AbstractString...)
```

Update all the files with paths specified by `files` in the index `idx` (or the index of the `repo`). Match the state of each file in the index with the current state on disk, removing it if it has been removed on disk, or updating its entry in the object database.

`LibGit2.url` – Function.

```
url(rmt::GitRemote)
```

Get the fetch URL of a remote git repository.

Examples

```
julia> repo_url = "https://github.com/JuliaLang/Example.jl";

julia> repo = LibGit2.init(mktempdir());

julia> remote = LibGit2.GitRemote(repo, "origin", repo_url);

julia> LibGit2.url(remote)
"https://github.com/JuliaLang/Example.jl"
```

`LibGit2.version` – Function.

```
version()::VersionNumber
```

Return the version of libgit2 in use, as a `VersionNumber`.

`LibGit2.with` – Function.

```
with(f::Function, obj)
```

Resource management helper function. Applies `f` to `obj`, making sure to call `close` on `obj` after `f` successfully returns or throws an error. Ensures that allocated git resources are finalized as soon as they are no longer needed.

`LibGit2.with_warn` – Function.

```
with_warn(f::Function, ::Type{T}, args...)
```

Resource management helper function. Apply `f` to `args`, first constructing an instance of type `T` from `args`. Makes sure to call `close` on the resulting object after `f` successfully returns or throws an error. Ensures that allocated git resources are finalized as soon as they are no longer needed. If an error is thrown by `f`, a warning is shown containing the error.

`LibGit2.workdir` – Function.

```
LibGit2.workdir(repo::GitRepo)
```

Return the location of the working directory of `repo`. This will throw an error for bare repositories.

> **Note**
>
> This will typically be the parent directory of `gitdir(repo)`, but can be different in some cases: e.g. if either the `core.worktree` configuration variable or the `GIT_WORK_TREE` environment variable is set.

See also `gitdir`, `path`.

`LibGit2.GitObject` – Method.

```
(::Type{T})(te::GitTreeEntry) where T<:GitObject
```

Get the git object to which `te` refers and return it as its actual type (the type `entrytype` would show), for instance a `GitBlob` or `GitTag`.

Examples

```
tree = LibGit2.GitTree(repo, "HEAD^{tree}")
tree_entry = tree[1]
blob = LibGit2.GitBlob(tree_entry)
```

`LibGit2.UserPasswordCredential` – Type.

Credential that support only `user` and `password` parameters

`LibGit2.SSHCredential` – Type.

SSH credential type

`LibGit2.isfilled` – Function.

```
isfilled(cred::AbstractCredential)::Bool
```

Verifies that a credential is ready for use in authentication.

`LibGit2.CachedCredentials` – Type.

Caches credential information for re-use

`LibGit2.CredentialPayload` – Type.

```
LibGit2.CredentialPayload
```

Retains the state between multiple calls to the credential callback for the same URL. A `CredentialPayload` instance is expected to be `reset!` whenever it will be used with a different URL.

`LibGit2.approve` – Function.

```
approve(payload::CredentialPayload; shred::Bool=true) -> nothing
```

Store the `payload` credential for re-use in a future authentication. Should only be called when authentication was successful.

The `shred` keyword controls whether sensitive information in the payload credential field should be destroyed. Should only be set to `false` during testing.

`LibGit2.reject` – Function.

```
reject(payload::CredentialPayload; shred::Bool=true) -> nothing
```

Discard the `payload` credential from begin re-used in future authentication. Should only be called when authentication was unsuccessful.

The `shred` keyword controls whether sensitive information in the payload credential field should be destroyed. Should only be set to `false` during testing.

`LibGit2.Consts.GIT_CONFIG` – Type.

Priority level of a config file.

These priority levels correspond to the natural escalation logic (from higher to lower) when searching for config entries in git.

- `CONFIG_LEVEL_DEFAULT` - Open the global, XDG and system configuration files if any available.
- `CONFIG_LEVEL_PROGRAMDATA` - System-wide on Windows, for compatibility with portable git
- `CONFIG_LEVEL_SYSTEM` - System-wide configuration file; `/etc/gitconfig` on Linux systems
- `CONFIG_LEVEL_XDG` - XDG compatible configuration file; typically `~/.config/git/config`
- `CONFIG_LEVEL_GLOBAL` - User-specific configuration file (also called Global configuration file); typically `~/.gitconfig`
- `CONFIG_LEVEL_LOCAL` - Repository specific configuration file; `$WORK_DIR/.git/config` on non-bare repos
- `CONFIG_LEVEL_WORKTREE` - Worktree specific configuration file; `$GIT_DIR/config.worktree`
- `CONFIG_LEVEL_APP` - Application specific configuration file; freely defined by applications
- `CONFIG_HIGHEST_LEVEL` - Represents the highest level available config file (i.e. the most specific config file available that actually is loaded)

Chapter 83

Linear Algebra

In addition to (and as part of) its support for multi-dimensional arrays, Julia provides native implementations of many common and useful linear algebra operations which can be loaded with `using LinearAlgebra`. Basic operations, such as `tr`, `det`, and `inv` are all supported:

```
julia> A = [1 2 3; 4 1 6; 7 8 1]
3×3 Matrix{Int64}:
 1  2  3
 4  1  6
 7  8  1

julia> tr(A)
3

julia> det(A)
104.0

julia> inv(A)
3×3 Matrix{Float64}:
 -0.451923   0.211538    0.0865385
  0.365385  -0.192308    0.0576923
  0.240385   0.0576923  -0.0673077
```

As well as other useful operations, such as finding eigenvalues or eigenvectors:

```
julia> A = [-4. -17.; 2. 2.]
2×2 Matrix{Float64}:
 -4.0  -17.0
  2.0    2.0

julia> eigvals(A)
2-element Vector{ComplexF64}:
 -1.0 - 5.0im
 -1.0 + 5.0im

julia> eigvecs(A)
2×2 Matrix{ComplexF64}:
```

```
  0.945905-0.0im        0.945905+0.0im
 -0.166924+0.278207im  -0.166924-0.278207im
```

In addition, Julia provides many factorizations which can be used to speed up problems such as linear solve or matrix exponentiation by pre-factorizing a matrix into a form more amenable (for performance or memory reasons) to the problem. See the documentation on `factorize` for more information. As an example:

```
julia> A = [1.5 2 -4; 3 -1 -6; -10 2.3 4]
3×3 Matrix{Float64}:
   1.5   2.0  -4.0
   3.0  -1.0  -6.0
 -10.0   2.3   4.0

julia> factorize(A)
LU{Float64, Matrix{Float64}, Vector{Int64}}
L factor:
3×3 Matrix{Float64}:
  1.0    0.0       0.0
 -0.15   1.0       0.0
 -0.3   -0.132196  1.0
U factor:
3×3 Matrix{Float64}:
 -10.0  2.3     4.0
   0.0  2.345  -3.4
   0.0  0.0    -5.24947
```

Since A is not Hermitian, symmetric, triangular, tridiagonal, or bidiagonal, an LU factorization may be the best we can do. Compare with:

```
julia> B = [1.5 2 -4; 2 -1 -3; -4 -3 5]
3×3 Matrix{Float64}:
  1.5   2.0  -4.0
  2.0  -1.0  -3.0
 -4.0  -3.0   5.0

julia> factorize(B)
BunchKaufman{Float64, Matrix{Float64}, Vector{Int64}}
D factor:
3×3 Tridiagonal{Float64, Vector{Float64}}:
 -1.64286   0.0   ⋅
  0.0      -2.8  0.0
   ⋅        0.0  5.0
U factor:
3×3 UnitUpperTriangular{Float64, Matrix{Float64}}:
 1.0  0.142857  -0.8
  ⋅   1.0       -0.6
  ⋅    ⋅         1.0
permutation:
3-element Vector{Int64}:
 1
```

```
 2
 3
```

Here, Julia was able to detect that B is in fact symmetric, and used a more appropriate factorization. Often it's possible to write more efficient code for a matrix that is known to have certain properties e.g. it is symmetric, or tridiagonal. Julia provides some special types so that you can "tag" matrices as having these properties. For instance:

```
julia> B = [1.5 2 -4; 2 -1 -3; -4 -3 5]
3×3 Matrix{Float64}:
  1.5   2.0  -4.0
  2.0  -1.0  -3.0
 -4.0  -3.0   5.0

julia> sB = Symmetric(B)
3×3 Symmetric{Float64, Matrix{Float64}}:
  1.5   2.0  -4.0
  2.0  -1.0  -3.0
 -4.0  -3.0   5.0
```

sB has been tagged as a matrix that's (real) symmetric, so for later operations we might perform on it, such as eigenfactorization or computing matrix-vector products, efficiencies can be found by only referencing half of it. For example:

```
julia> B = [1.5 2 -4; 2 -1 -3; -4 -3 5]
3×3 Matrix{Float64}:
  1.5   2.0  -4.0
  2.0  -1.0  -3.0
 -4.0  -3.0   5.0

julia> sB = Symmetric(B)
3×3 Symmetric{Float64, Matrix{Float64}}:
  1.5   2.0  -4.0
  2.0  -1.0  -3.0
 -4.0  -3.0   5.0

julia> x = [1; 2; 3]
3-element Vector{Int64}:
 1
 2
 3

julia> sB\x
3-element Vector{Float64}:
 -1.7391304347826084
 -1.1086956521739126
 -1.4565217391304346
```

The \ operation here performs the linear solution. The left-division operator is pretty powerful and it's easy to write compact, readable code that is flexible enough to solve all sorts of systems of linear equations.

83.1 Special matrices

Matrices with special symmetries and structures arise often in linear algebra and are frequently associated with various matrix factorizations. Julia features a rich collection of special matrix types, which allow for fast computation with specialized routines that are specially developed for particular matrix types.

The following tables summarize the types of special matrices that have been implemented in Julia, as well as whether hooks to various optimized methods for them in LAPACK are available.

Type	Description
`Symmetric`	Symmetric matrix
`Hermitian`	Hermitian matrix
`UpperTriangular`	Upper triangular matrix
`UnitUpperTriangular`	Upper triangular matrix with unit diagonal
`LowerTriangular`	Lower triangular matrix
`UnitLowerTriangular`	Lower triangular matrix with unit diagonal
`UpperHessenberg`	Upper Hessenberg matrix
`Tridiagonal`	Tridiagonal matrix
`SymTridiagonal`	Symmetric tridiagonal matrix
`Bidiagonal`	Upper/lower bidiagonal matrix
`Diagonal`	Diagonal matrix
`UniformScaling`	Uniform scaling operator

Elementary operations

Matrix type	+	-	*	\	Other functions with optimized methods
`Symmetric`				MV	`inv`, `sqrt`, `cbrt`, `exp`
`Hermitian`				MV	`inv`, `sqrt`, `cbrt`, `exp`
`UpperTriangular`			MV	MV	`inv`, `det`, `logdet`
`UnitUpperTriangular`			MV	MV	`inv`, `det`, `logdet`
`LowerTriangular`			MV	MV	`inv`, `det`, `logdet`
`UnitLowerTriangular`			MV	MV	`inv`, `det`, `logdet`
`UpperHessenberg`				MM	`inv`, `det`
`SymTridiagonal`	M	M	MS	MV	`eigmax`, `eigmin`
`Tridiagonal`	M	M	MS	MV	
`Bidiagonal`	M	M	MS	MV	
`Diagonal`	M	M	MV	MV	`inv`, `det`, `logdet`, `/`
`UniformScaling`	M	M	MVS	MVS	`/`

Legend:

Key	Description
M (matrix)	An optimized method for matrix-matrix operations is available
V (vector)	An optimized method for matrix-vector operations is available
S (scalar)	An optimized method for matrix-scalar operations is available

Matrix factorizations

Matrix type	LAPACK	eigen	eigvals	eigvecs	svd	svdvals
Symmetric	SY		ARI			
Hermitian	HE		ARI			
UpperTriangular	TR	A	A	A		
UnitUpperTriangular	TR	A	A	A		
LowerTriangular	TR	A	A	A		
UnitLowerTriangular	TR	A	A	A		
SymTridiagonal	ST	A	ARI	AV		
Tridiagonal	GT					
Bidiagonal	BD				A	A
Diagonal	DI		A			

Legend:

Key	Description	Example
A (all)	An optimized method to find all the characteristic values and/or vectors is available	e.g. `eigvals(M)`
R (range)	An optimized method to find the `il`th through the `ih`th characteristic values are available	`eigvals(M, il, ih)`
I (interval)	An optimized method to find the characteristic values in the interval `[vl, vh]` is available	`eigvals(M, vl, vh)`
V (vectors)	An optimized method to find the characteristic vectors corresponding to the characteristic values `x=[x1, x2,...]` is available	`eigvecs(M, x)`

The uniform scaling operator

A `UniformScaling` operator represents a scalar times the identity operator, `λ*I`. The identity operator `I` is defined as a constant and is an instance of `UniformScaling`. The size of these operators are generic and match the other matrix in the binary operations `+`, `-`, `*` and `\`. For `A+I` and `A-I` this means that `A` must be square. Multiplication with the identity operator `I` is a noop (except for checking that the scaling factor is one) and therefore almost without overhead.

To see the `UniformScaling` operator in action:

```
julia> U = UniformScaling(2);

julia> a = [1 2; 3 4]
2×2 Matrix{Int64}:
 1  2
 3  4

julia> a + U
2×2 Matrix{Int64}:
 3  2
 3  6
```

```
julia> a * U
2×2 Matrix{Int64}:
 2  4
 6  8

julia> [a U]
2×4 Matrix{Int64}:
 1  2  2  0
 3  4  0  2

julia> b = [1 2 3; 4 5 6]
2×3 Matrix{Int64}:
 1  2  3
 4  5  6

julia> b - U
ERROR: DimensionMismatch: matrix is not square: dimensions are (2, 3)
Stacktrace:
[...]
```

If you need to solve many systems of the form `(A+μI)x = b` for the same `A` and different `μ`, it might be beneficial to first compute the Hessenberg factorization `F` of `A` via the `hessenberg` function. Given `F`, Julia employs an efficient algorithm for `(F+μ*I) \ b` (equivalent to `(A+μ*I)x \ b`) and related operations like determinants.

83.2 Matrix factorizations

Matrix factorizations (a.k.a. matrix decompositions) compute the factorization of a matrix into a product of matrices, and are one of the central concepts in (numerical) linear algebra.

The following table summarizes the types of matrix factorizations that have been implemented in Julia. Details of their associated methods can be found in the Standard functions section of the Linear Algebra documentation.

Adjoints and transposes of `Factorization` objects are lazily wrapped in `AdjointFactorization` and `TransposeFactorization` objects, respectively. Generically, transpose of real `Factorizations` are wrapped as `AdjointFactorization`.

83.3 Orthogonal matrices (`AbstractQ`)

Some matrix factorizations generate orthogonal/unitary "matrix" factors. These factorizations include QR-related factorizations obtained from calls to `qr`, i.e., `QR`, `QRCompactWY` and `QRPivoted`, the Hessenberg factorization obtained from calls to `hessenberg`, and the LQ factorization obtained from `lq`. While these orthogonal/unitary factors admit a matrix representation, their internal representation is, for performance and memory reasons, different. Hence, they should be rather viewed as matrix-backed, function-based linear operators. In particular, reading, for instance, a column of its matrix representation requires running "matrix"-vector multiplication code, rather than simply reading out data from memory (possibly filling parts of the vector with structural zeros). Another clear distinction from other, non-triangular matrix types is that the underlying multiplication code allows for in-place modification during multiplication. Furthermore, objects of specific `AbstractQ` subtypes as those created via `qr`, `hessenberg` and `lq` can behave like a square or a rectangular matrix depending on context:

Type	Description
`BunchKaufman`	Bunch-Kaufman factorization
`Cholesky`	Cholesky factorization
`CholeskyPivoted`	Pivoted Cholesky factorization
`LDLt`	LDL(T) factorization
`LU`	LU factorization
`QR`	QR factorization
`QRCompactWY`	Compact WY form of the QR factorization
`QRPivoted`	Pivoted QR factorization
`LQ`	QR factorization of `transpose(A)`
`Hessenberg`	Hessenberg decomposition
`Eigen`	Spectral decomposition
`GeneralizedEigen`	Generalized spectral decomposition
`SVD`	Singular value decomposition
`GeneralizedSVD`	Generalized SVD
`Schur`	Schur decomposition
`GeneralizedSchur`	Generalized Schur decomposition

```
julia> using LinearAlgebra

julia> Q = qr(rand(3,2)).Q
3×3 LinearAlgebra.QRCompactWYQ{Float64, Matrix{Float64}, Matrix{Float64}}

julia> Matrix(Q)
3×2 Matrix{Float64}:
 -0.320597   0.865734
 -0.765834  -0.475694
 -0.557419   0.155628

julia> Q*I
3×3 Matrix{Float64}:
 -0.320597   0.865734  -0.384346
 -0.765834  -0.475694  -0.432683
 -0.557419   0.155628   0.815514

julia> Q*ones(2)
3-element Vector{Float64}:
  0.5451367118802273
 -1.241527373086654
 -0.40179067589600226

julia> Q*ones(3)
3-element Vector{Float64}:
  0.16079054743832022
 -1.674209978965636
  0.41372375588835797

julia> ones(1,2) * Q'
1×3 Matrix{Float64}:
 0.545137  -1.24153  -0.401791
```

```
julia> ones(1,3) * Q'
1×3 Matrix{Float64}:
 0.160791  -1.67421  0.413724
```

Due to this distinction from dense or structured matrices, the abstract `AbstractQ` type does not subtype `AbstractMatrix`, but instead has its own type hierarchy. Custom types that subtype `AbstractQ` can rely on generic fallbacks if the following interface is satisfied. For example, for

```
struct MyQ{T} <: LinearAlgebra.AbstractQ{T}
    # required fields
end
```

provide overloads for

```
Base.size(Q::MyQ) # size of corresponding square matrix representation
Base.convert(::Type{AbstractQ{T}}, Q::MyQ) # eltype promotion [optional]
LinearAlgebra.lmul!(Q::MyQ, x::AbstractVecOrMat) # left-multiplication
LinearAlgebra.rmul!(A::AbstractMatrix, Q::MyQ) # right-multiplication
```

If `eltype` promotion is not of interest, the `convert` method is unnecessary, since by default `convert(::Type{AbstractQ{T}}, Q::AbstractQ{T})` returns `Q` itself. Adjoints of `AbstractQ`-typed objects are lazily wrapped in an `AdjointQ` wrapper type, which requires its own `LinearAlgebra.lmul!` and `LinearAlgebra.rmul!` methods. Given this set of methods, any `Q::MyQ` can be used like a matrix, preferably in a multiplicative context: multiplication via `*` with scalars, vectors and matrices from left and right, obtaining a matrix representation of `Q` via `Matrix(Q)` (or `Q*I`) and indexing into the matrix representation all work. In contrast, addition and subtraction as well as more generally broadcasting over elements in the matrix representation fail because that would be highly inefficient. For such use cases, consider computing the matrix representation up front and cache it for future reuse.

83.4 Pivoting Strategies

Several of Julia's matrix factorizations support pivoting, which can be used to improve their numerical stability. In fact, some matrix factorizations, such as the LU factorization, may fail without pivoting.

In pivoting, first, a pivot element with good numerical properties is chosen based on a pivoting strategy. Next, the rows and columns of the original matrix are permuted to bring the chosen element in place for subsequent computation. Furthermore, the process is repeated for each stage of the factorization.

Consequently, besides the conventional matrix factors, the outputs of pivoted factorization schemes also include permutation matrices.

In the following, the pivoting strategies implemented in Julia are briefly described. Note that not all matrix factorizations may support them. Consult the documentation of the respective matrix factorization for details on the supported pivoting strategies.

See also `LinearAlgebra.ZeroPivotException`.

`LinearAlgebra.NoPivot` – Type.

```
    NoPivot
```

Pivoting is not performed. This is the default strategy for `cholesky` and `qr` factorizations. Note, however, that other matrix factorizations such as the LU factorization may fail without pivoting, and may also be numerically unstable for floating-point matrices in the face of roundoff error. In such cases, this pivot strategy is mainly useful for pedagogical purposes.

`LinearAlgebra.RowNonZero` – Type.

```
RowNonZero
```

First non-zero element in the remaining rows is chosen as the pivot element.

Beware that for floating-point matrices, the resulting LU algorithm is numerically unstable — this strategy is mainly useful for comparison to hand calculations (which typically use this strategy) or for other algebraic types (e.g. rational numbers) not susceptible to roundoff errors. Otherwise, the default `RowMaximum` pivoting strategy should be generally preferred in Gaussian elimination.

Note that the element type of the matrix must admit an `iszero` method.

`LinearAlgebra.RowMaximum` – Type.

```
RowMaximum
```

A row (and potentially also column) pivot is chosen based on a maximum property. This is the default strategy for LU factorization and for pivoted Cholesky factorization (though `[NoPivot]` is the default for `cholesky`).

In the LU case, the maximum-magnitude element within the current column in the remaining rows is chosen as the pivot element. This is sometimes referred to as the "partial pivoting" algorithm. In this case, the element type of the matrix must admit an `abs` method, whose result type must admit a `<` method.

In the Cholesky case, the maximal element among the remaining diagonal elements is chosen as the pivot element. This is sometimes referred to as the "diagonal pivoting" algorithm, and leads to *complete pivoting* (i.e., of both rows and columns by the same permutation). In this case, the (real part of the) element type of the matrix must admit a `<` method.

`LinearAlgebra.ColumnNorm` – Type.

```
ColumnNorm
```

The column with the maximum norm is used for subsequent computation. This is used for pivoted QR factorization.

Note that the element type of the matrix must admit `norm` and `abs` methods, whose respective result types must admit a `<` method.

83.5 Standard functions

Linear algebra functions in Julia are largely implemented by calling functions from LAPACK. Sparse matrix factorizations call functions from SuiteSparse. Other sparse solvers are available as Julia packages.

`Base.:*` – Method.

```
*(A::AbstractMatrix, B::AbstractMatrix)
```

Matrix multiplication.

Examples

```
julia> [1 1; 0 1] * [1 0; 1 1]
2×2 Matrix{Int64}:
 2  1
 1  1
```

Base.:* – Method.

```
*(A, B::AbstractMatrix, C)
A * B * C * D
```

Chained multiplication of 3 or 4 matrices is done in the most efficient sequence, based on the sizes of the arrays. That is, the number of scalar multiplications needed for `(A * B) * C` (with 3 dense matrices) is compared to that for `A * (B * C)` to choose which of these to execute.

If the last factor is a vector, or the first a transposed vector, then it is efficient to deal with these first. In particular `x' * B * y` means `(x' * B) * y` for an ordinary column-major `B::Matrix`. Unlike `dot(x, B, y)`, this allocates an intermediate array.

If the first or last factor is a number, this will be fused with the matrix multiplication, using 5-arg `mul!`.

See also `muladd`, `dot`.

Julia 1.7

These optimisations require at least Julia 1.7.

Base.:\ – Method.

```
\(A, B)
```

Matrix division using a polyalgorithm. For input matrices `A` and `B`, the result `X` is such that `A*X == B` when `A` is square. The solver that is used depends upon the structure of `A`. If `A` is upper or lower triangular (or diagonal), no factorization of `A` is required and the system is solved with either forward or backward substitution. For non-triangular square matrices, an LU factorization is used.

For rectangular `A` the result is the minimum-norm least squares solution computed by a pivoted QR factorization of `A` and a rank estimate of `A` based on the R factor.

When `A` is sparse, a similar polyalgorithm is used. For indefinite matrices, the `LDLt` factorization does not use pivoting during the numerical factorization and therefore the procedure can fail even for invertible matrices.

See also: `factorize`, `pinv`.

Examples

```
julia> A = [1 0; 1 -2]; B = [32; -4];

julia> X = A \ B
2-element Vector{Float64}:
 32.0
```

```
 18.0

julia> A * X == B
true
```

Base.:/ – Method.

```
A / B
```

Matrix right-division: `A / B` is equivalent to `(B' \ A')'` where `\` is the left-division operator. For square matrices, the result X is such that `A == X*B`.

See also: `rdiv!`.

Examples

```
julia> A = Float64[1 4 5; 3 9 2]; B = Float64[1 4 2; 3 4 2; 8 7 1];

julia> X = A / B
2×3 Matrix{Float64}:
 -0.65   3.75  -1.2
  3.25  -2.75   1.0

julia> isapprox(A, X*B)
true

julia> isapprox(X, A*pinv(B))
true
```

LinearAlgebra.SingularException – Type.

```
SingularException
```

Exception thrown when the input matrix has one or more zero-valued eigenvalues, and is not invertible. A linear solve involving such a matrix cannot be computed. The `info` field indicates the location of (one of) the singular value(s).

LinearAlgebra.PosDefException – Type.

```
PosDefException
```

Exception thrown when the input matrix was not positive definite. Some linear algebra functions and factorizations are only applicable to positive definite matrices. The `info` field indicates the location of (one of) the eigenvalue(s) which is (are) less than/equal to 0.

LinearAlgebra.ZeroPivotException – Type.

```
ZeroPivotException <: Exception
```

Exception thrown when a matrix factorization/solve encounters a zero in a pivot (diagonal) position and cannot proceed. This may *not* mean that the matrix is singular: it may be fruitful to switch to a different factorization such as pivoted LU that can re-order variables to eliminate spurious zero pivots. The `info` field indicates the location of (one of) the zero pivot(s).

`LinearAlgebra.RankDeficientException` – Type.

```
RankDeficientException
```

Exception thrown when the input matrix is rank deficient. Some linear algebra functions, such as the Cholesky decomposition, are only applicable to matrices that are not rank deficient. The `info` field indicates the computed rank of the matrix.

`LinearAlgebra.LAPACKException` – Type.

```
LAPACKException
```

Generic LAPACK exception thrown either during direct calls to the LAPACK functions or during calls to other functions that use the LAPACK functions internally but lack specialized error handling. The `info` field contains additional information on the underlying error and depends on the LAPACK function that was invoked.

`LinearAlgebra.dot` – Function.

```
dot(x, y)
x · y
```

Compute the dot product between two vectors. For complex vectors, the first vector is conjugated.

`dot` also works on arbitrary iterable objects, including arrays of any dimension, as long as `dot` is defined on the elements.

`dot` is semantically equivalent to `sum(dot(vx,vy) for (vx,vy) in zip(x, y))`, with the added restriction that the arguments must have equal lengths.

`x · y` (where `·` can be typed by tab-completing `\cdot` in the REPL) is a synonym for `dot(x, y)`.

Examples

```
julia> dot([1; 1], [2; 3])
5

julia> dot([im; im], [1; 1])
0 - 2im

julia> dot(1:5, 2:6)
70

julia> x = fill(2., (5,5));

julia> y = fill(3., (5,5));

julia> dot(x, y)
150.0
```

`LinearAlgebra.dot` – Method.

```
dot(x, A, y)
```

Compute the generalized dot product `dot(x, A*y)` between two vectors x and y, without storing the intermediate result of `A*y`. As for the two-argument `dot(_,_)`, this acts recursively. Moreover, for complex vectors, the first vector is conjugated.

Julia 1.4

Three-argument `dot` requires at least Julia 1.4.

Examples

```
julia> dot([1; 1], [1 2; 3 4], [2; 3])
26

julia> dot(1:5, reshape(1:25, 5, 5), 2:6)
4850

julia> ⋅(1:5, reshape(1:25, 5, 5), 2:6) == dot(1:5, reshape(1:25, 5, 5), 2:6)
true
```

`LinearAlgebra.cross` – Function.

```
cross(x, y)
×(x,y)
```

Compute the cross product of two 3-vectors.

Examples

```
julia> a = [0;1;0]
3-element Vector{Int64}:
 0
 1
 0

julia> b = [0;0;1]
3-element Vector{Int64}:
 0
 0
 1

julia> cross(a,b)
3-element Vector{Int64}:
 1
 0
 0
```

`LinearAlgebra.axpy!` – Function.

```
axpy!(α, x::AbstractArray, y::AbstractArray)
```

Overwrite y with `x * α + y` and return y. If x and y have the same axes, it's equivalent with `y .+= x .* a`.

Examples

```
julia> x = [1; 2; 3];

julia> y = [4; 5; 6];

julia> axpy!(2, x, y)
3-element Vector{Int64}:
  6
  9
 12
```

`LinearAlgebra.axpby!` – Function.

```
axpby!(α, x::AbstractArray, β, y::AbstractArray)
```

Overwrite y with `x * α + y * β` and return y. If x and y have the same axes, it's equivalent with `y .= x .* a .+ y .* β`.

Examples

```
julia> x = [1; 2; 3];

julia> y = [4; 5; 6];

julia> axpby!(2, x, 2, y)
3-element Vector{Int64}:
 10
 14
 18
```

`LinearAlgebra.rotate!` – Function.

```
rotate!(x, y, c, s)
```

Overwrite x with `s*y + c*x` and y with `c*y - conj(s)*x`. Returns x and y.

Julia 1.5

`rotate!` requires at least Julia 1.5.

`LinearAlgebra.reflect!` – Function.

```
reflect!(x, y, c, s)
```

Overwrite x with `c*x + s*y` and y with `conj(s)*x - c*y`. Returns x and y.

Julia 1.5

`reflect!` requires at least Julia 1.5.

`LinearAlgebra.factorize` – Function.

```
factorize(A)
```

Compute a convenient factorization of A, based upon the type of the input matrix. If A is passed as a generic matrix, `factorize` checks to see if it is symmetric/triangular/etc. To this end, `factorize` may check every element of A to verify/rule out each property. It will short-circuit as soon as it can rule out symmetry/triangular structure. The return value can be reused for efficient solving of multiple systems. For example: `A=factorize(A); x=A\b; y=A\C`.

Properties of A	type of factorization
Dense Symmetric/Hermitian	Bunch-Kaufman (see `bunchkaufman`)
Sparse Symmetric/Hermitian	LDLt (see `ldlt`)
Triangular	Triangular
Diagonal	Diagonal
Bidiagonal	Bidiagonal
Tridiagonal	LU (see `lu`)
Symmetric real tridiagonal	LDLt (see `ldlt`)
General square	LU (see `lu`)
General non-square	QR (see `qr`)

Examples

```
julia> A = Array(Bidiagonal(fill(1.0, (5, 5)), :U))
5×5 Matrix{Float64}:
 1.0  1.0  0.0  0.0  0.0
 0.0  1.0  1.0  0.0  0.0
 0.0  0.0  1.0  1.0  0.0
 0.0  0.0  0.0  1.0  1.0
 0.0  0.0  0.0  0.0  1.0

julia> factorize(A) # factorize will check to see that A is already factorized
5×5 Bidiagonal{Float64, Vector{Float64}}:
 1.0  1.0   ·    ·    ·
  ·   1.0  1.0   ·    ·
  ·    ·   1.0  1.0   ·
  ·    ·    ·   1.0  1.0
  ·    ·    ·    ·   1.0
```

This returns a `5×5 Bidiagonal{Float64}`, which can now be passed to other linear algebra functions (e.g. eigensolvers) which will use specialized methods for `Bidiagonal` types.

`LinearAlgebra.Diagonal` – Type.

```
Diagonal(V::AbstractVector)
```

Construct a lazy matrix with V as its diagonal.

See also `UniformScaling` for the lazy identity matrix `I`, `diagm` to make a dense matrix, and `diag` to extract diagonal elements.

Examples

```
julia> d = Diagonal([1, 10, 100])
3×3 Diagonal{Int64, Vector{Int64}}:
 1   ·    ·
```

```
 ·  10    ·
 ·   ·  100

julia> diagm([7, 13])
2×2 Matrix{Int64}:
 7   0
 0  13

julia> ans + I
2×2 Matrix{Int64}:
 8   0
 0  14

julia> I(2)
2×2 Diagonal{Bool, Vector{Bool}}:
 1  ·
 ·  1
```

Note

A one-column matrix is not treated like a vector, but instead calls the method `Diagonal(A::AbstractMatrix)` which extracts 1-element `diag(A)`:

```
julia> A = transpose([7.0 13.0])
2×1 transpose(::Matrix{Float64}) with eltype Float64:
  7.0
 13.0

julia> Diagonal(A)
1×1 Diagonal{Float64, Vector{Float64}}:
 7.0
```

```
Diagonal(A::AbstractMatrix)
```

Construct a matrix from the principal diagonal of `A`. The input matrix `A` may be rectangular, but the output will be square.

Examples

```
julia> A = [1 2; 3 4]
2×2 Matrix{Int64}:
 1  2
 3  4

julia> D = Diagonal(A)
2×2 Diagonal{Int64, Vector{Int64}}:
 1  ·
 ·  4

julia> A = [1 2 3; 4 5 6]
2×3 Matrix{Int64}:
 1  2  3
```

```
 4  5  6

julia> Diagonal(A)
2×2 Diagonal{Int64, Vector{Int64}}:
 1  ·
 ·  5
```

```
Diagonal{T}(undef, n)
```

Construct an uninitialized `Diagonal{T}` of length `n`. See `undef`.

`LinearAlgebra.Bidiagonal` – Type.

```
Bidiagonal(dv::V, ev::V, uplo::Symbol) where V <: AbstractVector
```

Constructs an upper (`uplo=:U`) or lower (`uplo=:L`) bidiagonal matrix using the given diagonal (`dv`) and off-diagonal (`ev`) vectors. The result is of type `Bidiagonal` and provides efficient specialized linear solvers, but may be converted into a regular matrix with `convert(Array, _)` (or `Array(_)` for short). The length of `ev` must be one less than the length of `dv`.

Examples

```
julia> dv = [1, 2, 3, 4]
4-element Vector{Int64}:
 1
 2
 3
 4

julia> ev = [7, 8, 9]
3-element Vector{Int64}:
 7
 8
 9

julia> Bu = Bidiagonal(dv, ev, :U) # ev is on the first superdiagonal
4×4 Bidiagonal{Int64, Vector{Int64}}:
 1  7  ·  ·
 ·  2  8  ·
 ·  ·  3  9
 ·  ·  ·  4

julia> Bl = Bidiagonal(dv, ev, :L) # ev is on the first subdiagonal
4×4 Bidiagonal{Int64, Vector{Int64}}:
 1  ·  ·  ·
 7  2  ·  ·
 ·  8  3  ·
 ·  ·  9  4
```

```
Bidiagonal(A, uplo::Symbol)
```

Construct a `Bidiagonal` matrix from the main diagonal of `A` and its first super- (if `uplo=:U`) or sub-diagonal (if `uplo=:L`).

Examples

```
julia> A = [1 1 1 1; 2 2 2 2; 3 3 3 3; 4 4 4 4]
4×4 Matrix{Int64}:
 1  1  1  1
 2  2  2  2
 3  3  3  3
 4  4  4  4

julia> Bidiagonal(A, :U) # contains the main diagonal and first superdiagonal of A
4×4 Bidiagonal{Int64, Vector{Int64}}:
 1  1  ⋅  ⋅
 ⋅  2  2  ⋅
 ⋅  ⋅  3  3
 ⋅  ⋅  ⋅  4

julia> Bidiagonal(A, :L) # contains the main diagonal and first subdiagonal of A
4×4 Bidiagonal{Int64, Vector{Int64}}:
 1  ⋅  ⋅  ⋅
 2  2  ⋅  ⋅
 ⋅  3  3  ⋅
 ⋅  ⋅  4  4
```

`LinearAlgebra.SymTridiagonal` – Type.

```
SymTridiagonal(dv::V, ev::V) where V <: AbstractVector
```

Construct a symmetric tridiagonal matrix from the diagonal (dv) and first sub/super-diagonal (ev), respectively. The result is of type `SymTridiagonal` and provides efficient specialized eigensolvers, but may be converted into a regular matrix with `convert(Array, _)` (or `Array(_)` for short).

For `SymTridiagonal` block matrices, the elements of `dv` are symmetrized. The argument `ev` is interpreted as the superdiagonal. Blocks from the subdiagonal are (materialized) transpose of the corresponding superdiagonal blocks.

Examples

```
julia> dv = [1, 2, 3, 4]
4-element Vector{Int64}:
 1
 2
 3
 4

julia> ev = [7, 8, 9]
3-element Vector{Int64}:
 7
 8
 9

julia> SymTridiagonal(dv, ev)
4×4 SymTridiagonal{Int64, Vector{Int64}}:
 1  7  ⋅  ⋅
 7  2  8  ⋅
 ⋅  8  3  9
 ⋅  ⋅  9  4
```

```
julia> A = SymTridiagonal(fill([1 2; 3 4], 3), fill([1 2; 3 4], 2));

julia> A[1,1]
2×2 Symmetric{Int64, Matrix{Int64}}:
 1  2
 2  4

julia> A[1,2]
2×2 Matrix{Int64}:
 1  2
 3  4

julia> A[2,1]
2×2 Matrix{Int64}:
 1  3
 2  4
```

```
SymTridiagonal(A::AbstractMatrix)
```

Construct a symmetric tridiagonal matrix from the diagonal and first superdiagonal of the symmetric matrix `A`.

Examples

```
julia> A = [1 2 3; 2 4 5; 3 5 6]
3×3 Matrix{Int64}:
 1  2  3
 2  4  5
 3  5  6

julia> SymTridiagonal(A)
3×3 SymTridiagonal{Int64, Vector{Int64}}:
 1  2  ·
 2  4  5
 ·  5  6

julia> B = reshape([[1 2; 2 3], [1 2; 3 4], [1 3; 2 4], [1 2; 2 3]], 2, 2);

julia> SymTridiagonal(B)
2×2 SymTridiagonal{Matrix{Int64}, Vector{Matrix{Int64}}}:
 [1 2; 2 3]  [1 3; 2 4]
 [1 2; 3 4]  [1 2; 2 3]
```

`LinearAlgebra.Tridiagonal` – Type.

```
Tridiagonal(dl::V, d::V, du::V) where V <: AbstractVector
```

Construct a tridiagonal matrix from the first subdiagonal, diagonal, and first superdiagonal, respectively. The result is of type `Tridiagonal` and provides efficient specialized linear solvers, but may be converted into a regular matrix with `convert(Array, _)` (or `Array(_)` for short). The lengths of `dl` and `du` must be one less than the length of `d`.

Note

The subdiagonal `dl` and the superdiagonal `du` must not be aliased to each other. If aliasing is detected, the constructor will use a copy of `du` as its argument.

Examples

```
julia> dl = [1, 2, 3];

julia> du = [4, 5, 6];

julia> d = [7, 8, 9, 0];

julia> Tridiagonal(dl, d, du)
4×4 Tridiagonal{Int64, Vector{Int64}}:
 7  4  ·  ·
 1  8  5  ·
 ·  2  9  6
 ·  ·  3  0
```

```
Tridiagonal(A)
```

Construct a tridiagonal matrix from the first sub-diagonal, diagonal and first super-diagonal of the matrix `A`.

Examples

```
julia> A = [1 2 3 4; 1 2 3 4; 1 2 3 4; 1 2 3 4]
4×4 Matrix{Int64}:
 1  2  3  4
 1  2  3  4
 1  2  3  4
 1  2  3  4

julia> Tridiagonal(A)
4×4 Tridiagonal{Int64, Vector{Int64}}:
 1  2  ·  ·
 1  2  3  ·
 ·  2  3  4
 ·  ·  3  4
```

`LinearAlgebra.Symmetric` – Type.

```
Symmetric(A::AbstractMatrix, uplo::Symbol=:U)
```

Construct a `Symmetric` view of the upper (if `uplo = :U`) or lower (if `uplo = :L`) triangle of the matrix `A`.

`Symmetric` views are mainly useful for real-symmetric matrices, for which specialized algorithms (e.g. for eigenproblems) are enabled for `Symmetric` types. More generally, see also `Hermitian(A)` for Hermitian matrices `A == A'`, which is effectively equivalent to `Symmetric` for real matrices but is also useful for complex matrices. (Whereas complex `Symmetric` matrices are supported but have few if any specialized algorithms.)

To compute the symmetric part of a real matrix, or more generally the Hermitian part `(A + A') / 2` of a real or complex matrix `A`, use `hermitianpart`.

The `uplo` symbol corresponding to the triangular half of `A` that is shared by the symmetric view may be fetched by using the function `LinearAlgebra.uplo`. The underlying matrix `A` may be fetched from the symmetric view by using `parent`.

Examples

```
julia> A = [1 2 3; 4 5 6; 7 8 9]
3×3 Matrix{Int64}:
 1  2  3
 4  5  6
 7  8  9

julia> Supper = Symmetric(A)
3×3 Symmetric{Int64, Matrix{Int64}}:
 1  2  3
 2  5  6
 3  6  9

julia> Slower = Symmetric(A, :L)
3×3 Symmetric{Int64, Matrix{Int64}}:
 1  4  7
 4  5  8
 7  8  9

julia> hermitianpart(A)
3×3 Hermitian{Float64, Matrix{Float64}}:
 1.0  3.0  5.0
 3.0  5.0  7.0
 5.0  7.0  9.0
```

Note that `Supper` will not be equal to `Slower` unless `A` is itself symmetric (e.g. if `A == transpose(A)`).

`LinearAlgebra.Hermitian` – Type.

```
Hermitian(A::AbstractMatrix, uplo::Symbol=:U)
```

Construct a `Hermitian` view of the upper (if `uplo = :U`) or lower (if `uplo = :L`) triangle of the matrix `A`.

To compute the Hermitian part of `A`, use `hermitianpart`.

The `uplo` symbol corresponding to the triangular half of `A` that is shared by the hermitian view may be fetched by using the function `LinearAlgebra.uplo`. The underlying matrix `A` may be fetched from the hermitian view by using `parent`.

Examples

```
julia> A = [1 2+2im 3-3im; 4 5 6-6im; 7 8+8im 9]
3×3 Matrix{Complex{Int64}}:
 1+0im  2+2im  3-3im
 4+0im  5+0im  6-6im
 7+0im  8+8im  9+0im
```

```
julia> Hupper = Hermitian(A)
3×3 Hermitian{Complex{Int64}, Matrix{Complex{Int64}}}:
 1+0im  2+2im  3-3im
 2-2im  5+0im  6-6im
 3+3im  6+6im  9+0im

julia> Hlower = Hermitian(A, :L)
3×3 Hermitian{Complex{Int64}, Matrix{Complex{Int64}}}:
 1+0im  4+0im  7+0im
 4+0im  5+0im  8-8im
 7+0im  8+8im  9+0im

julia> hermitianpart(A)
3×3 Hermitian{ComplexF64, Matrix{ComplexF64}}:
 1.0+0.0im  3.0+1.0im  5.0-1.5im
 3.0-1.0im  5.0+0.0im  7.0-7.0im
 5.0+1.5im  7.0+7.0im  9.0+0.0im
```

Note that `Hupper` will not be equal to `Hlower` unless `A` is itself Hermitian (e.g. if `A == adjoint(A)`).

All non-real parts of the diagonal will be ignored.

```
Hermitian(fill(complex(1,1), 1, 1)) == fill(1, 1, 1)
```

`LinearAlgebra.LowerTriangular` – Type.

```
LowerTriangular(A::AbstractMatrix)
```

Construct a `LowerTriangular` view of the matrix `A`.

Examples

```
julia> A = [1.0 2.0 3.0; 4.0 5.0 6.0; 7.0 8.0 9.0]
3×3 Matrix{Float64}:
 1.0  2.0  3.0
 4.0  5.0  6.0
 7.0  8.0  9.0

julia> LowerTriangular(A)
3×3 LowerTriangular{Float64, Matrix{Float64}}:
 1.0   ⋅    ⋅
 4.0  5.0   ⋅
 7.0  8.0  9.0
```

`LinearAlgebra.UpperTriangular` – Type.

```
UpperTriangular(A::AbstractMatrix)
```

Construct an `UpperTriangular` view of the matrix `A`.

Examples

```
julia> A = [1.0 2.0 3.0; 4.0 5.0 6.0; 7.0 8.0 9.0]
3×3 Matrix{Float64}:
 1.0  2.0  3.0
 4.0  5.0  6.0
 7.0  8.0  9.0

julia> UpperTriangular(A)
3×3 UpperTriangular{Float64, Matrix{Float64}}:
 1.0  2.0  3.0
  ·   5.0  6.0
  ·    ·   9.0
```

`LinearAlgebra.UnitLowerTriangular` – Type.

```
UnitLowerTriangular(A::AbstractMatrix)
```

Construct a `UnitLowerTriangular` view of the matrix `A`. Such a view has the `oneunit` of the `eltype` of `A` on its diagonal.

Examples

```
julia> A = [1.0 2.0 3.0; 4.0 5.0 6.0; 7.0 8.0 9.0]
3×3 Matrix{Float64}:
 1.0  2.0  3.0
 4.0  5.0  6.0
 7.0  8.0  9.0

julia> UnitLowerTriangular(A)
3×3 UnitLowerTriangular{Float64, Matrix{Float64}}:
 1.0   ·    ·
 4.0  1.0   ·
 7.0  8.0  1.0
```

`LinearAlgebra.UnitUpperTriangular` – Type.

```
UnitUpperTriangular(A::AbstractMatrix)
```

Construct an `UnitUpperTriangular` view of the matrix `A`. Such a view has the `oneunit` of the `eltype` of `A` on its diagonal.

Examples

```
julia> A = [1.0 2.0 3.0; 4.0 5.0 6.0; 7.0 8.0 9.0]
3×3 Matrix{Float64}:
 1.0  2.0  3.0
 4.0  5.0  6.0
 7.0  8.0  9.0

julia> UnitUpperTriangular(A)
3×3 UnitUpperTriangular{Float64, Matrix{Float64}}:
 1.0  2.0  3.0
  ·   1.0  6.0
  ·    ·   1.0
```

`LinearAlgebra.UpperHessenberg` – Type.

```
UpperHessenberg(A::AbstractMatrix)
```

Construct an `UpperHessenberg` view of the matrix `A`. Entries of `A` below the first subdiagonal are ignored.

Julia 1.3

This type was added in Julia 1.3.

Efficient algorithms are implemented for `H \ b`, `det(H)`, and similar.

See also the `hessenberg` function to factor any matrix into a similar upper-Hessenberg matrix.

If `F::Hessenberg` is the factorization object, the unitary matrix can be accessed with `F.Q` and the Hessenberg matrix with `F.H`. When `Q` is extracted, the resulting type is the `HessenbergQ` object, and may be converted to a regular matrix with `convert(Array, _)` (or `Array(_)` for short).

Iterating the decomposition produces the factors `F.Q` and `F.H`.

Examples

```
julia> A = [1 2 3 4; 5 6 7 8; 9 10 11 12; 13 14 15 16]
4×4 Matrix{Int64}:
  1   2   3   4
  5   6   7   8
  9  10  11  12
 13  14  15  16

julia> UpperHessenberg(A)
4×4 UpperHessenberg{Int64, Matrix{Int64}}:
 1   2   3   4
 5   6   7   8
 ·  10  11  12
 ·   ·  15  16
```

`LinearAlgebra.UniformScaling` – Type.

```
UniformScaling{T<:Number}
```

Generically sized uniform scaling operator defined as a scalar times the identity operator, `λ*I`. Although without an explicit `size`, it acts similarly to a matrix in many cases and includes support for some indexing. See also `I`.

Julia 1.6

Indexing using ranges is available as of Julia 1.6.

Examples

```
julia> J = UniformScaling(2.)
UniformScaling{Float64}
2.0*I
```

```
julia> A = [1. 2.; 3. 4.]
2×2 Matrix{Float64}:
 1.0  2.0
 3.0  4.0

julia> J*A
2×2 Matrix{Float64}:
 2.0  4.0
 6.0  8.0

julia> J[1:2, 1:2]
2×2 Matrix{Float64}:
 2.0  0.0
 0.0  2.0
```

`LinearAlgebra.I` – Constant.

```
I
```

An object of type `UniformScaling`, representing an identity matrix of any size.

Examples

```
julia> fill(1, (5,6)) * I == fill(1, (5,6))
true

julia> [1 2im 3; 1im 2 3] * I
2×3 Matrix{Complex{Int64}}:
 1+0im  0+2im  3+0im
 0+1im  2+0im  3+0im
```

`LinearAlgebra.UniformScaling` – Method.

```
(I::UniformScaling)(n::Integer)
```

Construct a `Diagonal` matrix from a `UniformScaling`.

Julia 1.2

This method is available as of Julia 1.2.

Examples

```
julia> I(3)
3×3 Diagonal{Bool, Vector{Bool}}:
 1  ·  ·
 ·  1  ·
 ·  ·  1

julia> (0.7*I)(3)
3×3 Diagonal{Float64, Vector{Float64}}:
 0.7   ·    ·
  ·   0.7   ·
  ·    ·   0.7
```

`LinearAlgebra.Factorization` – Type.

```
LinearAlgebra.Factorization
```

Abstract type for matrix factorizations a.k.a. matrix decompositions. See online documentation for a list of available matrix factorizations.

`LinearAlgebra.LU` – Type.

```
LU <: Factorization
```

Matrix factorization type of the LU factorization of a square matrix `A`. This is the return type of `lu`, the corresponding matrix factorization function.

The individual components of the factorization `F::LU` can be accessed via `getproperty`:

Component	Description
`F.L`	L (unit lower triangular) part of LU
`F.U`	U (upper triangular) part of LU
`F.p`	(right) permutation `Vector`
`F.P`	(right) permutation `Matrix`

Iterating the factorization produces the components `F.L`, `F.U`, and `F.p`.

Examples

```
julia> A = [4 3; 6 3]
2×2 Matrix{Int64}:
 4  3
 6  3

julia> F = lu(A)
LU{Float64, Matrix{Float64}, Vector{Int64}}
L factor:
2×2 Matrix{Float64}:
 1.0       0.0
 0.666667  1.0
U factor:
2×2 Matrix{Float64}:
 6.0  3.0
 0.0  1.0

julia> F.L * F.U == A[F.p, :]
true

julia> l, u, p = lu(A); # destructuring via iteration

julia> l == F.L && u == F.U && p == F.p
true
```

`LinearAlgebra.lu` – Function.

```
lu(A::AbstractSparseMatrixCSC; check = true, q = nothing, control = get_umfpack_control()) ->
↪  F::UmfpackLU
```

Compute the LU factorization of a sparse matrix A.

For sparse A with real or complex element type, the return type of F is UmfpackLU{Tv, Ti}, with Tv = Float64 or ComplexF64 respectively and Ti is an integer type (Int32 or Int64).

When check = true, an error is thrown if the decomposition fails. When check = false, responsibility for checking the decomposition's validity (via issuccess) lies with the user.

The permutation q can either be a permutation vector or nothing. If no permutation vector is provided or q is nothing, UMFPACK's default is used. If the permutation is not zero-based, a zero-based copy is made.

The control vector defaults to the Julia SparseArrays package's default configuration for UMFPACK (NB: this is modified from the UMFPACK defaults to disable iterative refinement), but can be changed by passing a vector of length UMFPACK_CONTROL, see the UMFPACK manual for possible configurations. For example to reenable iterative refinement:

```
umfpack_control = SparseArrays.UMFPACK.get_umfpack_control(Float64, Int64) # read Julia
↪  default configuration for a Float64 sparse matrix
SparseArrays.UMFPACK.show_umf_ctrl(umfpack_control) # optional - display values
umfpack_control[SparseArrays.UMFPACK.JL_UMFPACK_IRSTEP] = 2.0 # reenable iterative refinement
↪  (2 is UMFPACK default max iterative refinement steps)

Alu = lu(A; control = umfpack_control)
x = Alu \ b   # solve Ax = b, including UMFPACK iterative refinement
```

The individual components of the factorization F can be accessed by indexing:

Component	Description
L	L (lower triangular) part of LU
U	U (upper triangular) part of LU
p	right permutation Vector
q	left permutation Vector
Rs	Vector of scaling factors
:	(L,U,p,q,Rs) components

The relation between F and A is

F.L*F.U == (F.Rs .* A)[F.p, F.q]

F further supports the following functions:

- \
- det

See also lu!

Note

lu(A::AbstractSparseMatrixCSC) uses the UMFPACK[1] library that is part of SuiteSparse. As this library only supports sparse matrices with Float64 or ComplexF64 elements, lu converts A into a copy that is of type SparseMatrixCSC{Float64} or SparseMatrixCSC{ComplexF64} as appropriate.

```
lu(A, pivot = RowMaximum(); check = true, allowsingular = false) -> F::LU
```

Compute the LU factorization of `A`.

When `check = true`, an error is thrown if the decomposition fails. When `check = false`, responsibility for checking the decomposition's validity (via `issuccess`) lies with the user.

By default, with `check = true`, an error is also thrown when the decomposition produces valid factors, but the upper-triangular factor `U` is rank-deficient. This may be changed by passing `allowsingular = true`.

In most cases, if `A` is a subtype `S` of `AbstractMatrix{T}` with an element type `T` supporting `+`, `-`, `*` and `/`, the return type is `LU{T,S{T}}`.

In general, LU factorization involves a permutation of the rows of the matrix (corresponding to the `F.p` output described below), known as "pivoting" (because it corresponds to choosing which row contains the "pivot", the diagonal entry of `F.U`). One of the following pivoting strategies can be selected via the optional `pivot` argument:

- `RowMaximum()` (default): the standard pivoting strategy; the pivot corresponds to the element of maximum absolute value among the remaining, to be factorized rows. This pivoting strategy requires the element type to also support `abs` and `<`. (This is generally the only numerically stable option for floating-point matrices.)
- `RowNonZero()`: the pivot corresponds to the first non-zero element among the remaining, to be factorized rows. (This corresponds to the typical choice in hand calculations, and is also useful for more general algebraic number types that support `iszero` but not `abs` or `<`.)
- `NoPivot()`: pivoting turned off (will fail if a zero entry is encountered in a pivot position, even when `allowsingular = true`).

The individual components of the factorization `F` can be accessed via `getproperty`:

Component	Description
`F.L`	`L` (lower triangular) part of `LU`
`F.U`	`U` (upper triangular) part of `LU`
`F.p`	(right) permutation `Vector`
`F.P`	(right) permutation `Matrix`

Iterating the factorization produces the components `F.L`, `F.U`, and `F.p`.

The relationship between `F` and `A` is

```
F.L*F.U == A[F.p, :]
```

`F` further supports the following functions:

Julia 1.11

The `allowsingular` keyword argument was added in Julia 1.11.

Examples

[1]Davis, Timothy A. (2004b). Algorithm 832: UMFPACK V4.3—an Unsymmetric-Pattern Multifrontal Method. ACM Trans. Math. Softw., 30(2), 196–199. doi:10.1145/992200.992206

Supported function	LU	LU{T,Tridiagonal{T}}
/	✓	
\	✓	✓
inv	✓	✓
det	✓	✓
logdet	✓	✓
logabsdet	✓	✓
size	✓	✓

```
julia> A = [4 3; 6 3]
2×2 Matrix{Int64}:
 4  3
 6  3

julia> F = lu(A)
LU{Float64, Matrix{Float64}, Vector{Int64}}
L factor:
2×2 Matrix{Float64}:
 1.0       0.0
 0.666667  1.0
U factor:
2×2 Matrix{Float64}:
 6.0  3.0
 0.0  1.0

julia> F.L * F.U == A[F.p, :]
true

julia> l, u, p = lu(A); # destructuring via iteration

julia> l == F.L && u == F.U && p == F.p
true

julia> lu([1 2; 1 2], allowsingular = true)
LU{Float64, Matrix{Float64}, Vector{Int64}}
L factor:
2×2 Matrix{Float64}:
 1.0  0.0
 1.0  1.0
U factor (rank-deficient):
2×2 Matrix{Float64}:
 1.0  2.0
 0.0  0.0
```

`LinearAlgebra.lu!` – Function.

```
lu!(F::UmfpackLU, A::AbstractSparseMatrixCSC; check=true, reuse_symbolic=true, q=nothing) ->
↪  F::UmfpackLU
```

Compute the LU factorization of a sparse matrix `A`, reusing the symbolic factorization of an already existing LU factorization stored in `F`. Unless `reuse_symbolic` is set to false, the sparse matrix `A` must

have an identical nonzero pattern as the matrix used to create the LU factorization F, otherwise an error is thrown. If the size of A and F differ, all vectors will be resized accordingly.

When `check = true`, an error is thrown if the decomposition fails. When `check = false`, responsibility for checking the decomposition's validity (via `issuccess`) lies with the user.

The permutation `q` can either be a permutation vector or `nothing`. If no permutation vector is provided or `q` is `nothing`, UMFPACK's default is used. If the permutation is not zero based, a zero based copy is made.

See also `lu`

Note

`lu!(F::UmfpackLU, A::AbstractSparseMatrixCSC)` uses the UMFPACK library that is part of SuiteSparse. As this library only supports sparse matrices with `Float64` or `ComplexF64` elements, `lu!` will automatically convert the types to those set by the LU factorization or `SparseMatrixCSC{ComplexF64}` as appropriate.

Julia 1.5

`lu!` for `UmfpackLU` requires at least Julia 1.5.

Examples

```
julia> A = sparse(Float64[1.0 2.0; 0.0 3.0]);

julia> F = lu(A);

julia> B = sparse(Float64[1.0 1.0; 0.0 1.0]);

julia> lu!(F, B);

julia> F \ ones(2)
2-element Vector{Float64}:
 0.0
 1.0
```

```
lu!(F::LU, pivot = RowMaximum(); check = true, allowsingular = false) -> LU
```

`lu!` is the same as `lu`, but saves space by overwriting the input F, instead of creating a copy.

Julia 1.13

Reusing dense LU factorizations in `lu!` require Julia 1.13 or later.

Examples

```
julia> A = [4. 3.; 6. 3.]
2×2 Matrix{Float64}:
 4.0  3.0
 6.0  3.0
```

```
julia> F = lu(A)
LU{Float64, Matrix{Float64}, Vector{Int64}}
L factor:
2×2 Matrix{Float64}:
 1.0       0.0
 0.666667  1.0
U factor:
2×2 Matrix{Float64}:
 6.0  3.0
 0.0  1.0

julia> B = [8 3; 12 3]
2×2 Matrix{Int64}:
  8  3
 12  3

julia> F2 = lu!(F,B)
LU{Float64, Matrix{Float64}, Vector{Int64}}
L factor:
2×2 Matrix{Float64}:
 1.0       0.0
 0.666667  1.0
U factor:
2×2 Matrix{Float64}:
 12.0  3.0
  0.0  1.0
```

```
lu!(A, pivot = RowMaximum(); check = true, allowsingular = false) -> LU
```

`lu!` is the same as `lu`, but saves space by overwriting the input `A`, instead of creating a copy. An `InexactError` exception is thrown if the factorization produces a number not representable by the element type of `A`, e.g. for integer types.

Julia 1.11

The `allowsingular` keyword argument was added in Julia 1.11.

Examples

```
julia> A = [4. 3.; 6. 3.]
2×2 Matrix{Float64}:
 4.0  3.0
 6.0  3.0

julia> F = lu!(A)
LU{Float64, Matrix{Float64}, Vector{Int64}}
L factor:
2×2 Matrix{Float64}:
 1.0       0.0
 0.666667  1.0
U factor:
2×2 Matrix{Float64}:
 6.0  3.0
```

```
 0.0  1.0

julia> iA = [4 3; 6 3]
2×2 Matrix{Int64}:
 4  3
 6  3

julia> lu!(iA)
ERROR: InexactError: Int64(0.6666666666666666)
Stacktrace:
[...]
```

`LinearAlgebra.Cholesky` – Type.

```
Cholesky <: Factorization
```

Matrix factorization type of the Cholesky factorization of a dense symmetric/Hermitian positive definite matrix `A`. This is the return type of `cholesky`, the corresponding matrix factorization function.

The triangular Cholesky factor can be obtained from the factorization `F::Cholesky` via `F.L` and `F.U`, where `A ≈ F.U' * F.U ≈ F.L * F.L'`.

The following functions are available for `Cholesky` objects: `size`, `\`, `inv`, `det`, `logdet` and `isposdef`.

Iterating the decomposition produces the components `L` and `U`.

Examples

```
julia> A = [4. 12. -16.; 12. 37. -43.; -16. -43. 98.]
3×3 Matrix{Float64}:
   4.0   12.0  -16.0
  12.0   37.0  -43.0
 -16.0  -43.0   98.0

julia> C = cholesky(A)
Cholesky{Float64, Matrix{Float64}}
U factor:
3×3 UpperTriangular{Float64, Matrix{Float64}}:
 2.0  6.0  -8.0
  ·   1.0   5.0
  ·    ·    3.0

julia> C.U
3×3 UpperTriangular{Float64, Matrix{Float64}}:
 2.0  6.0  -8.0
  ·   1.0   5.0
  ·    ·    3.0

julia> C.L
3×3 LowerTriangular{Float64, Adjoint{Float64, Matrix{Float64}}}:
  2.0   ·    ·
  6.0  1.0   ·
 -8.0  5.0  3.0

julia> C.L * C.U == A
```

```
true

julia> l, u = C; # destructuring via iteration

julia> l == C.L && u == C.U
true
```

`LinearAlgebra.CholeskyPivoted` – Type.

```
CholeskyPivoted
```

Matrix factorization type of the pivoted Cholesky factorization of a dense symmetric/Hermitian positive semi-definite matrix `A`. This is the return type of `cholesky(_, ::RowMaximum)`, the corresponding matrix factorization function.

The triangular Cholesky factor can be obtained from the factorization `F::CholeskyPivoted` via `F.L` and `F.U`, and the permutation via `F.p`, where `A[F.p, F.p] ≈ Ur' * Ur ≈ Lr * Lr'` with `Ur = F.U[1:F.rank, :]` and `Lr = F.L[:, 1:F.rank]`, or alternatively `A ≈ Up' * Up ≈ Lp * Lp'` with `Up = F.U[1:F.rank, invperm(F.p)]` and `Lp = F.L[invperm(F.p), 1:F.rank]`.

The following functions are available for `CholeskyPivoted` objects: `size`, `\`, `inv`, `det`, and `rank`.

Iterating the decomposition produces the components `L` and `U`.

Examples

```
julia> X = [1.0, 2.0, 3.0, 4.0];

julia> A = X * X';

julia> C = cholesky(A, RowMaximum(), check = false)
CholeskyPivoted{Float64, Matrix{Float64}, Vector{Int64}}
U factor with rank 1:
4×4 UpperTriangular{Float64, Matrix{Float64}}:
 4.0  2.0  3.0  1.0
  ·   0.0  6.0  2.0
  ·    ·   9.0  3.0
  ·    ·    ·   1.0
permutation:
4-element Vector{Int64}:
 4
 2
 3
 1

julia> C.U[1:C.rank, :]' * C.U[1:C.rank, :] ≈ A[C.p, C.p]
true

julia> l, u = C; # destructuring via iteration

julia> l == C.L && u == C.U
true
```

`LinearAlgebra.cholesky` – Function.

```
cholesky(A::SparseMatrixCSC; shift = 0.0, check = true, perm = nothing) -> CHOLMOD.Factor
```

Compute the Cholesky factorization of a sparse positive definite matrix `A`. `A` must be a `SparseMatrixCSC` or a `Symmetric`/`Hermitian` view of a `SparseMatrixCSC`. Note that if `A` doesn't have the type tag, it must itself be symmetric or Hermitian. If `perm` is not given, a fill-reducing permutation is used. `F = cholesky(A)` is most frequently used to solve systems of equations with `F\b`, but also the methods `diag`, `det`, and `logdet` are defined for `F`. You can also extract individual factors from `F`, using `F.L`. However, since pivoting is on by default, the factorization is internally represented as `A == P'*L*L'*P` with a permutation matrix `P`; using just `L` without accounting for `P` will give incorrect answers. To include the effects of permutation, it's typically preferable to extract "combined" factors like `PtL = F.PtL` (the equivalent of `P'*L`) and `LtP = F.UP` (the equivalent of `L'*P`). The complete list of supported factors is `:L, :PtL, :UP, :U`. The permutation vector is available as `F.p`, defined such that `L*L' == A[p, p]`,

The `L` component can be materialized as a sparse matrix using `sparse(F.L)`. Other components cannot be materialized directly, but can be reconstructed from `sparse(F.L)` and `F.p` if needed.

When `check = true`, an error is thrown if the decomposition fails. When `check = false`, responsibility for checking the decomposition's validity (via `issuccess`) lies with the user.

Setting the optional `shift` keyword argument computes the factorization of `A+shift*I` instead of `A`. If the `perm` argument is provided, it should be a permutation of `1:size(A,1)` giving the ordering to use (instead of CHOLMOD's default AMD ordering).

See also `ldlt` for a similar factorization that does not require positive definiteness, but can be significantly slower than `cholesky`.

Examples

In the following example, the fill-reducing permutation used is `[3, 2, 1]`. If `perm` is set to `1:3` to enforce no permutation, the number of nonzero elements in the factor is 6.

```
julia> A = [2 1 1; 1 2 0; 1 0 2]
3×3 Matrix{Int64}:
 2  1  1
 1  2  0
 1  0  2

julia> C = cholesky(sparse(A))
SparseArrays.CHOLMOD.Factor{Float64, Int64}
type:    LLt
method:  simplicial
maxnnz:  5
nnz:     5
success: true

julia> C.p
3-element Vector{Int64}:
 3
 2
 1

julia> L = sparse(C.L);

julia> Matrix(L)
```

```
3×3 Matrix{Float64}:
 1.41421   0.0       0.0
 0.0       1.41421   0.0
 0.707107  0.707107  1.0

julia> L * L' ≈ A[C.p, C.p]
true

julia> P = sparse(1:3, C.p, ones(3))
3×3 SparseMatrixCSC{Float64, Int64} with 3 stored entries:
  ·    ·   1.0
  ·   1.0   ·
 1.0   ·    ·

julia> P' * L * L' * P ≈ A
true

julia> C = cholesky(sparse(A), perm=1:3)
SparseArrays.CHOLMOD.Factor{Float64, Int64}
type:    LLt
method:  simplicial
maxnnz:  6
nnz:     6
success: true

julia> L = sparse(C.L);

julia> Matrix(L)
3×3 Matrix{Float64}:
 1.41421    0.0       0.0
 0.707107   1.22474   0.0
 0.707107  -0.408248  1.1547

julia> L * L' ≈ A
true
```

Note

This method uses the CHOLMOD[2,3] library from SuiteSparse. CHOLMOD only supports real or complex types in single or double precision. Input matrices not of those element types will be converted to these types as appropriate.

Many other functions from CHOLMOD are wrapped but not exported from the `Base.SparseArrays.CHOLMOD` module.

```
cholesky(A, NoPivot(); check = true) -> Cholesky
```

[2] Chen, Y., Davis, T. A., Hager, W. W., & Rajamanickam, S. (2008). Algorithm 887: CHOLMOD, Supernodal Sparse Cholesky Factorization and Update/Downdate. ACM Trans. Math. Softw., 35(3). doi:10.1145/1391989.1391995

[3] Davis, Timothy A., & Hager, W. W. (2009). Dynamic Supernodes in Sparse Cholesky Update/Downdate and Triangular Solves. ACM Trans. Math. Softw., 35(4). doi:10.1145/1462173.1462176

Compute the Cholesky factorization of a dense symmetric positive definite matrix `A` and return a `Cholesky` factorization. The matrix `A` can either be a `Symmetric` or `Hermitian AbstractMatrix` or a *perfectly* symmetric or Hermitian `AbstractMatrix`.

The triangular Cholesky factor can be obtained from the factorization `F` via `F.L` and `F.U`, where `A ≈ F.U' * F.U ≈ F.L * F.L'`.

The following functions are available for `Cholesky` objects: `size`, `\`, `inv`, `det`, `logdet` and `isposdef`.

If you have a matrix `A` that is slightly non-Hermitian due to roundoff errors in its construction, wrap it in `Hermitian(A)` before passing it to `cholesky` in order to treat it as perfectly Hermitian.

When `check = true`, an error is thrown if the decomposition fails. When `check = false`, responsibility for checking the decomposition's validity (via `issuccess`) lies with the user.

Examples

```
julia> A = [4. 12. -16.; 12. 37. -43.; -16. -43. 98.]
3×3 Matrix{Float64}:
   4.0   12.0  -16.0
  12.0   37.0  -43.0
 -16.0  -43.0   98.0

julia> C = cholesky(A)
Cholesky{Float64, Matrix{Float64}}
U factor:
3×3 UpperTriangular{Float64, Matrix{Float64}}:
 2.0  6.0  -8.0
  ·   1.0   5.0
  ·    ·    3.0

julia> C.U
3×3 UpperTriangular{Float64, Matrix{Float64}}:
 2.0  6.0  -8.0
  ·   1.0   5.0
  ·    ·    3.0

julia> C.L
3×3 LowerTriangular{Float64, Adjoint{Float64, Matrix{Float64}}}:
  2.0   ·    ·
  6.0  1.0   ·
 -8.0  5.0  3.0

julia> C.L * C.U == A
true
```

```
cholesky(A, RowMaximum(); tol = 0.0, check = true) -> CholeskyPivoted
```

Compute the pivoted Cholesky factorization of a dense symmetric positive semi-definite matrix `A` and return a `CholeskyPivoted` factorization. The matrix `A` can either be a `Symmetric` or `Hermitian AbstractMatrix` or a *perfectly* symmetric or Hermitian `AbstractMatrix`.

The triangular Cholesky factor can be obtained from the factorization `F` via `F.L` and `F.U`, and the permutation via `F.p`, where `A[F.p, F.p] ≈ Ur' * Ur ≈ Lr * Lr'` with `Ur = F.U[1:F.rank, :]` and `Lr =`

`F.L[:, 1:F.rank]`, or alternatively `A ≈ Up' * Up ≈ Lp * Lp'` with `Up = F.U[1:F.rank, invperm(F.p)]` and `Lp = F.L[invperm(F.p), 1:F.rank]`.

The following functions are available for `CholeskyPivoted` objects: `size`, `\`, `inv`, `det`, and `rank`.

The argument `tol` determines the tolerance for determining the rank. For negative values, the tolerance is equal to `eps()*size(A,1)*maximum(diag(A))`.

If you have a matrix `A` that is slightly non-Hermitian due to roundoff errors in its construction, wrap it in `Hermitian(A)` before passing it to `cholesky` in order to treat it as perfectly Hermitian.

When `check = true`, an error is thrown if the decomposition fails. When `check = false`, responsibility for checking the decomposition's validity (via `issuccess`) lies with the user.

Examples

```
julia> X = [1.0, 2.0, 3.0, 4.0];

julia> A = X * X';

julia> C = cholesky(A, RowMaximum(), check = false)
CholeskyPivoted{Float64, Matrix{Float64}, Vector{Int64}}
U factor with rank 1:
4×4 UpperTriangular{Float64, Matrix{Float64}}:
 4.0  2.0  3.0  1.0
  ·   0.0  6.0  2.0
  ·    ·   9.0  3.0
  ·    ·    ·   1.0
permutation:
4-element Vector{Int64}:
 4
 2
 3
 1

julia> C.U[1:C.rank, :]' * C.U[1:C.rank, :] ≈ A[C.p, C.p]
true

julia> l, u = C; # destructuring via iteration

julia> l == C.L && u == C.U
true
```

`LinearAlgebra.cholesky!` – Function.

```
cholesky!(F::CHOLMOD.Factor, A::SparseMatrixCSC; shift = 0.0, check = true) -> CHOLMOD.Factor
```

Compute the Cholesky (LL') factorization of `A`, reusing the symbolic factorization `F`. `A` must be a `SparseMatrixCSC` or a `Symmetric`/`Hermitian` view of a `SparseMatrixCSC`. Note that if `A` doesn't have the type tag, it must itself be symmetric or Hermitian.

See also `cholesky`.

Note

This method uses the CHOLMOD library from SuiteSparse, which only supports real or complex types in single or double precision. Input matrices not of those element types will be converted to these types as appropriate.

```
cholesky!(A::AbstractMatrix, NoPivot(); check = true) -> Cholesky
```

The same as `cholesky`, but saves space by overwriting the input `A`, instead of creating a copy. An `InexactError` exception is thrown if the factorization produces a number not representable by the element type of `A`, e.g. for integer types.

Examples

```
julia> A = [1 2; 2 50]
2×2 Matrix{Int64}:
 1   2
 2  50

julia> cholesky!(A)
ERROR: InexactError: Int64(6.782329983125268)
Stacktrace:
[...]
```

```
cholesky!(A::AbstractMatrix, RowMaximum(); tol = 0.0, check = true) -> CholeskyPivoted
```

The same as `cholesky`, but saves space by overwriting the input `A`, instead of creating a copy. An `InexactError` exception is thrown if the factorization produces a number not representable by the element type of `A`, e.g. for integer types.

`LinearAlgebra.lowrankupdate` – Function.

```
lowrankupdate(F::CHOLMOD.Factor, C::AbstractArray) -> FF::CHOLMOD.Factor
```

Get an `LDLt` Factorization of `A + C*C'` given an `LDLt` or `LLt` factorization `F` of `A`.

The returned factor is always an `LDLt` factorization.

See also `lowrankupdate!`, `lowrankdowndate`, `lowrankdowndate!`.

```
lowrankupdate(C::Cholesky, v::AbstractVector) -> CC::Cholesky
```

Update a Cholesky factorization `C` with the vector `v`. If `A = C.U'C.U` then `CC = cholesky(C.U'C.U + v*v')` but the computation of `CC` only uses `O(n^2)` operations.

`LinearAlgebra.lowrankdowndate` – Function.

```
lowrankdowndate(F::CHOLMOD.Factor, C::AbstractArray) -> FF::CHOLMOD.Factor
```

Get an `LDLt` Factorization of `A + C*C'` given an `LDLt` or `LLt` factorization `F` of `A`.

The returned factor is always an `LDLt` factorization.

See also `lowrankdowndate!`, `lowrankupdate`, `lowrankupdate!`.

```
lowrankdowndate(C::Cholesky, v::AbstractVector) -> CC::Cholesky
```

Downdate a Cholesky factorization C with the vector v. If A = C.U'C.U then CC = cholesky(C.U'C.U - v*v') but the computation of CC only uses O(n^2) operations.

LinearAlgebra.lowrankupdate! – Function.

```
lowrankupdate!(F::CHOLMOD.Factor, C::AbstractArray)
```

Update an LDLt or LLt Factorization F of A to a factorization of A + C*C'.

LLt factorizations are converted to LDLt.

See also lowrankupdate, lowrankdowndate, lowrankdowndate!.

```
lowrankupdate!(C::Cholesky, v::AbstractVector) -> CC::Cholesky
```

Update a Cholesky factorization C with the vector v. If A = C.U'C.U then CC = cholesky(C.U'C.U + v*v') but the computation of CC only uses O(n^2) operations. The input factorization C is updated in place such that on exit C == CC. The vector v is destroyed during the computation.

LinearAlgebra.lowrankdowndate! – Function.

```
lowrankdowndate!(F::CHOLMOD.Factor, C::AbstractArray)
```

Update an LDLt or LLt Factorization F of A to a factorization of A - C*C'.

LLt factorizations are converted to LDLt.

See also lowrankdowndate, lowrankupdate, lowrankupdate!.

```
lowrankdowndate!(C::Cholesky, v::AbstractVector) -> CC::Cholesky
```

Downdate a Cholesky factorization C with the vector v. If A = C.U'C.U then CC = cholesky(C.U'C.U - v*v') but the computation of CC only uses O(n^2) operations. The input factorization C is updated in place such that on exit C == CC. The vector v is destroyed during the computation.

LinearAlgebra.LDLt – Type.

```
LDLt <: Factorization
```

Matrix factorization type of the LDLt factorization of a real SymTridiagonal matrix S such that S = L*Diagonal(d)*L', where L is a UnitLowerTriangular matrix and d is a vector. The main use of an LDLt factorization F = ldlt(S) is to solve the linear system of equations Sx = b with F\b. This is the return type of ldlt, the corresponding matrix factorization function.

The individual components of the factorization F::LDLt can be accessed via getproperty:

Component	Description
F.L	L (unit lower triangular) part of LDLt
F.D	D (diagonal) part of LDLt
F.Lt	Lt (unit upper triangular) part of LDLt
F.d	diagonal values of D as a Vector

Examples

```
julia> S = SymTridiagonal([3., 4., 5.], [1., 2.])
3×3 SymTridiagonal{Float64, Vector{Float64}}:
 3.0  1.0   ·
 1.0  4.0  2.0
  ·   2.0  5.0

julia> F = ldlt(S)
LDLt{Float64, SymTridiagonal{Float64, Vector{Float64}}}
L factor:
3×3 UnitLowerTriangular{Float64, SymTridiagonal{Float64, Vector{Float64}}}:
 1.0       ·         ·
 0.333333  1.0       ·
 0.0       0.545455  1.0
D factor:
3×3 Diagonal{Float64, Vector{Float64}}:
 3.0   ·        ·
  ·   3.66667   ·
  ·    ·       3.90909
```

`LinearAlgebra.ldlt` – Function.

```
ldlt(A::SparseMatrixCSC; shift = 0.0, check = true, perm=nothing) -> CHOLMOD.Factor
```

Compute the LDL' factorization of a sparse matrix `A`. `A` must be a `SparseMatrixCSC` or a `Symmetric`/`Hermitian` view of a `SparseMatrixCSC`. Note that if `A` doesn't have the type tag, it must itself be symmetric or Hermitian. A fill-reducing permutation is used. `F = ldlt(A)` is most frequently used to solve systems of equations `A*x = b` with `F\b`. The returned factorization object `F` also supports the methods `diag`, `det`, `logdet`, and `inv`. You can extract individual factors from `F` using `F.L`. However, since pivoting is on by default, the factorization is internally represented as `A == P'*L*D*L'*P` with a permutation matrix `P`; using just `L` without accounting for `P` will give incorrect answers. To include the effects of permutation, it is typically preferable to extract "combined" factors like `PtL = F.PtL` (the equivalent of `P'*L`) and `LtP = F.UP` (the equivalent of `L'*P`). The complete list of supported factors is `:L, :PtL, :D, :UP, :U, :LD, :DU, :PtLD, :DUP`. The permutation vector is available as `F.p`, defined such that `L*D*L' == A[p, p]`.

The `LD` component can be materialized as a sparse matrix using `sparse(F.LD)`, Other components cannot be materialized directly, but can be reconstructed from `sparse(F.LD)` and `F.p` if needed.

Unlike the related Cholesky factorization, the LDL' factorization does not require `A` to be positive definite. However, it still requires all leading principal minors to be well-conditioned and will fail if this is not satisfied.

When `check = true`, an error is thrown if the decomposition fails. When `check = false`, responsibility for checking the decomposition's validity (via `issuccess`) lies with the user.

Setting the optional `shift` keyword argument computes the factorization of `A+shift*I` instead of `A`. If the `perm` argument is provided, it should be a permutation of `1:size(A,1)` giving the ordering to use (instead of CHOLMOD's default AMD ordering).

See also `cholesky` for a factorization that can be significantly faster than `ldlt`, but requires `A` to be positive definite.

Note

This method uses the CHOLMOD[2,3] library from SuiteSparse. CHOLMOD only supports real or complex types in single or double precision. Input matrices not of those element types will be converted to these types as appropriate.

Many other functions from CHOLMOD are wrapped but not exported from the `Base.SparseArrays.CHOLMOD` module.

```
ldlt(S::SymTridiagonal) -> LDLt
```

Compute an `LDLt` (i.e., LDL^T) factorization of the real symmetric tridiagonal matrix `S` such that `S = L*Diagonal(d)*L'` where `L` is a unit lower triangular matrix and `d` is a vector. The main use of an `LDLt` factorization `F = ldlt(S)` is to solve the linear system of equations `Sx = b` with `F\b`.

See also `bunchkaufman` for a similar, but pivoted, factorization of arbitrary symmetric or Hermitian matrices.

Examples

```
julia> S = SymTridiagonal([3., 4., 5.], [1., 2.])
3×3 SymTridiagonal{Float64, Vector{Float64}}:
 3.0  1.0   ⋅
 1.0  4.0  2.0
  ⋅   2.0  5.0

julia> ldltS = ldlt(S);

julia> b = [6., 7., 8.];

julia> ldltS \ b
3-element Vector{Float64}:
 1.7906976744186047
 0.627906976744186
 1.3488372093023255

julia> S \ b
3-element Vector{Float64}:
 1.7906976744186047
 0.627906976744186
 1.3488372093023255
```

`LinearAlgebra.ldlt!` – Function.

```
ldlt!(F::CHOLMOD.Factor, A::SparseMatrixCSC; shift = 0.0, check = true) -> CHOLMOD.Factor
```

Compute the LDL' factorization of `A`, reusing the symbolic factorization `F`. `A` must be a `SparseMatrixCSC` or a `Symmetric`/`Hermitian` view of a `SparseMatrixCSC`. Note that if `A` doesn't have the type tag, it must itself be symmetric or Hermitian.

See also `ldlt`.

Note

This method uses the CHOLMOD library from SuiteSparse, which only supports real or complex types in single or double precision. Input matrices not of those element types will be converted to these types as appropriate.

```
ldlt!(S::SymTridiagonal) -> LDLt
```

Same as `ldlt`, but saves space by overwriting the input `S`, instead of creating a copy.

Examples

```
julia> S = SymTridiagonal([3., 4., 5.], [1., 2.])
3×3 SymTridiagonal{Float64, Vector{Float64}}:
 3.0  1.0   ·
 1.0  4.0  2.0
  ·   2.0  5.0

julia> ldltS = ldlt!(S);

julia> ldltS === S
false

julia> S
3×3 SymTridiagonal{Float64, Vector{Float64}}:
 3.0       0.333333   ·
 0.333333  3.66667   0.545455
  ·        0.545455  3.90909
```

`LinearAlgebra.QR` – Type.

```
QR <: Factorization
```

A QR matrix factorization stored in a packed format, typically obtained from `qr`. If A is an `m×n` matrix, then

$$A = QR$$

where Q is an orthogonal/unitary matrix and R is upper triangular. The matrix Q is stored as a sequence of Householder reflectors v_i and coefficients τ_i where:

$$Q = \prod_{i=1}^{\min(m,n)} (I - \tau_i v_i v_i^T).$$

Iterating the decomposition produces the components `Q` and `R`.

The object has two fields:

- `factors` is an `m×n` matrix.

  - The upper triangular part contains the elements of R, that is `R = triu(F.factors)` for a `QR` object `F`.
  - The subdiagonal part contains the reflectors v_i stored in a packed format where v_i is the ith column of the matrix `V = I + tril(F.factors, -1)`.
- `τ` is a vector of length `min(m,n)` containing the coefficients τ_i.

`LinearAlgebra.QRCompactWY` – Type.

```
QRCompactWY <: Factorization
```

A QR matrix factorization stored in a compact blocked format, typically obtained from `qr`. If A is an `m×n` matrix, then

$$A = QR$$

where Q is an orthogonal/unitary matrix and R is upper triangular. It is similar to the `QR` format except that the orthogonal/unitary matrix Q is stored in *Compact WY* format [4]. For the block size n_b, it is stored as a `m×n` lower trapezoidal matrix V and a matrix $T = (T_1\ T_2\ \ldots\ T_{b-1}\ T'_b)$ composed of $b = \lceil \min(m,n)/n_b \rceil$ upper triangular matrices T_j of size $n_b \times n_b$ ($j = 1, \ldots, b-1$) and an upper trapezoidal $n_b \times \min(m,n) - (b-1)n_b$ matrix T'_b ($j = b$) whose upper square part denoted with T_b satisfying

$$Q = \prod_{i=1}^{\min(m,n)} (I - \tau_i v_i v_i^T) = \prod_{j=1}^{b} (I - V_j T_j V_j^T)$$

such that v_i is the ith column of V, τ_i is the ith element of `[diag(T_1); diag(T_2); …; diag(T_b)]`, and $(V_1\ V_2\ \ldots\ V_b)$ is the left `m×min(m, n)` block of V. When constructed using `qr`, the block size is given by $n_b = \min(m, n, 36)$.

Iterating the decomposition produces the components `Q` and `R`.

The object has two fields:

- `factors`, as in the `QR` type, is an `m×n` matrix.
  - The upper triangular part contains the elements of R, that is `R = triu(F.factors)` for a `QR` object `F`.
  - The subdiagonal part contains the reflectors v_i stored in a packed format such that `V = I + tril(F.factors, -1)`.
- `T` is a n_b-by-$\min(m,n)$ matrix as described above. The subdiagonal elements for each triangular matrix T_j are ignored.

Note

This format should not to be confused with the older *WY* representation [5].

`LinearAlgebra.QRPivoted` – Type.

```
QRPivoted <: Factorization
```

A QR matrix factorization with column pivoting in a packed format, typically obtained from `qr`. If A is an `m×n` matrix, then

$$AP = QR$$

where P is a permutation matrix, Q is an orthogonal/unitary matrix and R is upper triangular. The matrix Q is stored as a sequence of Householder reflectors:

$$Q = \prod_{i=1}^{\min(m,n)} (I - \tau_i v_i v_i^T).$$

Iterating the decomposition produces the components `Q`, `R`, and `p`.

The object has three fields:

- `factors` is an `m×n` matrix.
  - The upper triangular part contains the elements of R, that is `R = triu(F.factors)` for a `QR` object `F`.
  - The subdiagonal part contains the reflectors v_i stored in a packed format where v_i is the ith column of the matrix `V = I + tril(F.factors, -1)`.
- `τ` is a vector of length `min(m,n)` containing the coefficients au_i.
- `jpvt` is an integer vector of length `n` corresponding to the permutation P.

`LinearAlgebra.qr` – Function.

```
qr(A::SparseMatrixCSC; tol=_default_tol(A), ordering=ORDERING_DEFAULT) -> QRSparse
```

Compute the `QR` factorization of a sparse matrix `A`. Fill-reducing row and column permutations are used such that `F.R = F.Q'*A[F.prow,F.pcol]`. The main application of this type is to solve least squares or underdetermined problems with `\`. The function calls the C library SPQR[6].

Note

The returned `QRSparse` object uses an internal workspace for `ldiv!()` calls that is protected by a lock for threadsafety. For multithreaded use, create a separate copy of this object for each task with `copy(F)`.

[5] C Bischof and C Van Loan, "The WY representation for products of Householder matrices", SIAM J Sci Stat Comput 8 (1987), s2-s13. doi:10.1137/0908009

[4] R Schreiber and C Van Loan, "A storage-efficient WY representation for products of Householder transformations", SIAM J Sci Stat Comput 10 (1989), 53-57. doi:10.1137/0910005

Note

`qr(A::SparseMatrixCSC)` uses the SPQR library that is part of SuiteSparse. As this library only supports sparse matrices with `Float64` or `ComplexF64` elements, as of Julia v1.4 `qr` converts `A` into a copy that is of type `SparseMatrixCSC{Float64}` or `SparseMatrixCSC{ComplexF64}` as appropriate.

Examples

```
julia> A = sparse([1,2,3,4], [1,1,2,2], [1.0,1.0,1.0,1.0])
4×2 SparseMatrixCSC{Float64, Int64} with 4 stored entries:
 1.0   ·
 1.0   ·
  ·   1.0
  ·   1.0

julia> qr(A)
SparseArrays.SPQR.QRSparse{Float64, Int64}
Q factor:
4×4 SparseArrays.SPQR.QRSparseQ{Float64, Int64}
R factor:
2×2 SparseMatrixCSC{Float64, Int64} with 2 stored entries:
 -1.41421    ·
   ·       -1.41421
Row permutation:
4-element Vector{Int64}:
 1
 3
 4
 2
Column permutation:
2-element Vector{Int64}:
 1
 2
```

```
qr(A, pivot = NoPivot(); blocksize) -> F
```

Compute the QR factorization of the matrix `A`: an orthogonal (or unitary if `A` is complex-valued) matrix `Q`, and an upper triangular matrix `R` such that

$$A = QR$$

The returned object `F` stores the factorization in a packed format:

- if `pivot == ColumnNorm()` then `F` is a `QRPivoted` object,
- otherwise if the element type of `A` is a BLAS type (`Float32`, `Float64`, `ComplexF32` or `ComplexF64`), then `F` is a `QRCompactWY` object,

[6] Foster, L. V., & Davis, T. A. (2013). Algorithm 933: Reliable Calculation of Numerical Rank, Null Space Bases, Pseudoinverse Solutions, and Basic Solutions Using SuitesparseQR. ACM Trans. Math. Softw., 40(1). doi:10.1145/2513109.2513116

- otherwise F is a `QR` object.

The individual components of the decomposition F can be retrieved via property accessors:

- `F.Q`: the orthogonal/unitary matrix `Q`
- `F.R`: the upper triangular matrix `R`
- `F.p`: the permutation vector of the pivot (`QRPivoted` only)
- `F.P`: the permutation matrix of the pivot (`QRPivoted` only)

Note

Each reference to the upper triangular factor via `F.R` allocates a new array. It is therefore advisable to cache that array, say, by `R = F.R` and continue working with `R`.

Iterating the decomposition produces the components `Q`, `R`, and if extant `p`.

The following functions are available for the `QR` objects: `inv`, `size`, and `\`. When `A` is rectangular, `\` will return a least squares solution and if the solution is not unique, the one with smallest norm is returned. When `A` is not full rank, factorization with (column) pivoting is required to obtain a minimum norm solution.

Multiplication with respect to either full/square or non-full/square `Q` is allowed, i.e. both `F.Q*F.R` and `F.Q*A` are supported. A `Q` matrix can be converted into a regular matrix with `Matrix`. This operation returns the "thin" Q factor, i.e., if `A` is `m×n` with `m>=n`, then `Matrix(F.Q)` yields an `m×n` matrix with orthonormal columns. To retrieve the "full" Q factor, an `m×m` orthogonal matrix, use `F.Q*I` or `collect(F.Q)`. If `m<=n`, then `Matrix(F.Q)` yields an `m×m` orthogonal matrix.

The block size for QR decomposition can be specified by keyword argument `blocksize :: Integer` when `pivot == NoPivot()` and `A isa StridedMatrix{<:BlasFloat}`. It is ignored when `blocksize > minimum(size(A))`. See `QRCompactWY`.

Julia 1.4

The `blocksize` keyword argument requires Julia 1.4 or later.

Examples

```
julia> A = [3.0 -6.0; 4.0 -8.0; 0.0 1.0]
3×2 Matrix{Float64}:
 3.0  -6.0
 4.0  -8.0
 0.0   1.0

julia> F = qr(A)
LinearAlgebra.QRCompactWY{Float64, Matrix{Float64}, Matrix{Float64}}
Q factor: 3×3 LinearAlgebra.QRCompactWYQ{Float64, Matrix{Float64}, Matrix{Float64}}
R factor:
2×2 Matrix{Float64}:
 -5.0  10.0
  0.0  -1.0

julia> F.Q * F.R == A
true
```

Note

qr returns multiple types because LAPACK uses several representations that minimize the memory storage requirements of products of Householder elementary reflectors, so that the Q and R matrices can be stored compactly rather than two separate dense matrices.

`LinearAlgebra.qr!` – Function.

```
qr!(A, pivot = NoPivot(); blocksize)
```

`qr!` is the same as `qr` when `A` is a subtype of `AbstractMatrix`, but saves space by overwriting the input `A`, instead of creating a copy. An `InexactError` exception is thrown if the factorization produces a number not representable by the element type of `A`, e.g. for integer types.

Julia 1.4

The `blocksize` keyword argument requires Julia 1.4 or later.

Examples

```
julia> a = [1. 2.; 3. 4.]
2×2 Matrix{Float64}:
 1.0  2.0
 3.0  4.0

julia> qr!(a)
LinearAlgebra.QRCompactWY{Float64, Matrix{Float64}, Matrix{Float64}}
Q factor: 2×2 LinearAlgebra.QRCompactWYQ{Float64, Matrix{Float64}, Matrix{Float64}}
R factor:
2×2 Matrix{Float64}:
 -3.16228  -4.42719
  0.0      -0.632456

julia> a = [1 2; 3 4]
2×2 Matrix{Int64}:
 1  2
 3  4

julia> qr!(a)
ERROR: InexactError: Int64(3.1622776601683795)
Stacktrace:
[...]
```

`LinearAlgebra.LQ` – Type.

```
LQ <: Factorization
```

Matrix factorization type of the `LQ` factorization of a matrix `A`. The `LQ` decomposition is the `QR` decomposition of `transpose(A)`. This is the return type of `lq`, the corresponding matrix factorization function.

If `S::LQ` is the factorization object, the lower triangular component can be obtained via `S.L`, and the orthogonal/unitary component via `S.Q`, such that `A ≈ S.L*S.Q`.

Iterating the decomposition produces the components `S.L` and `S.Q`.

Examples

```
julia> A = [5. 7.; -2. -4.]
2×2 Matrix{Float64}:
  5.0   7.0
 -2.0  -4.0

julia> S = lq(A)
LQ{Float64, Matrix{Float64}, Vector{Float64}}
L factor:
2×2 Matrix{Float64}:
 -8.60233   0.0
  4.41741  -0.697486
Q factor: 2×2 LinearAlgebra.LQPackedQ{Float64, Matrix{Float64}, Vector{Float64}}

julia> S.L * S.Q
2×2 Matrix{Float64}:
  5.0   7.0
 -2.0  -4.0

julia> l, q = S; # destructuring via iteration

julia> l == S.L &&  q == S.Q
true
```

`LinearAlgebra.lq` – Function.

```
lq(A) -> S::LQ
```

Compute the LQ decomposition of `A`. The decomposition's lower triangular component can be obtained from the `LQ` object `S` via `S.L`, and the orthogonal/unitary component via `S.Q`, such that `A ≈ S.L*S.Q`.

Iterating the decomposition produces the components `S.L` and `S.Q`.

The LQ decomposition is the QR decomposition of `transpose(A)`, and it is useful in order to compute the minimum-norm solution `lq(A) \ b` to an underdetermined system of equations (`A` has more columns than rows, but has full row rank).

Examples

```
julia> A = [5. 7.; -2. -4.]
2×2 Matrix{Float64}:
  5.0   7.0
 -2.0  -4.0

julia> S = lq(A)
LQ{Float64, Matrix{Float64}, Vector{Float64}}
L factor:
2×2 Matrix{Float64}:
 -8.60233   0.0
  4.41741  -0.697486
Q factor: 2×2 LinearAlgebra.LQPackedQ{Float64, Matrix{Float64}, Vector{Float64}}

julia> S.L * S.Q
```

```
2×2 Matrix{Float64}:
  5.0   7.0
 -2.0  -4.0

julia> l, q = S; # destructuring via iteration

julia> l == S.L &&  q == S.Q
true
```

`LinearAlgebra.lq!` – Function.

```
lq!(A) -> LQ
```

Compute the `LQ` factorization of `A`, using the input matrix as a workspace. See also `lq`.

`LinearAlgebra.BunchKaufman` – Type.

```
BunchKaufman <: Factorization
```

Matrix factorization type of the Bunch-Kaufman factorization of a symmetric or Hermitian matrix `A` as `P'UDU'P` or `P'LDL'P`, depending on whether the upper (the default) or the lower triangle is stored in `A`. Here, `U` and `L` are respectively upper and lower triangular, `P` is a permutation matrix, and `D` is block diagonal with 1-by-1 and 2-by-2 blocks. If `A` is complex symmetric then `U'` and `L'` denote the unconjugated transposes, i.e. `transpose(U)` and `transpose(L)`, respectively. This is the return type of `bunchkaufman`, the corresponding matrix factorization function.

If `S::BunchKaufman` is the factorization object, the components can be obtained via `S.D`, `S.U` or `S.L` as appropriate given `S.uplo`, and `S.p`.

Iterating the decomposition produces the components `S.D`, `S.U` or `S.L` as appropriate given `S.uplo`, and `S.p`.

Examples

```
julia> A = Float64.([1 2; 2 3])
2×2 Matrix{Float64}:
 1.0  2.0
 2.0  3.0

julia> S = bunchkaufman(A) # A gets wrapped internally by Symmetric(A)
BunchKaufman{Float64, Matrix{Float64}, Vector{Int64}}
D factor:
2×2 Tridiagonal{Float64, Vector{Float64}}:
 -0.333333  0.0
  0.0       3.0
U factor:
2×2 UnitUpperTriangular{Float64, Matrix{Float64}}:
 1.0  0.666667
  ·   1.0
permutation:
2-element Vector{Int64}:
 1
 2
```

```
julia> d, u, p = S; # destructuring via iteration

julia> d == S.D && u == S.U && p == S.p
true

julia> S = bunchkaufman(Symmetric(A, :L))
BunchKaufman{Float64, Matrix{Float64}, Vector{Int64}}
D factor:
2×2 Tridiagonal{Float64, Vector{Float64}}:
 3.0   0.0
 0.0  -0.333333
L factor:
2×2 UnitLowerTriangular{Float64, Matrix{Float64}}:
 1.0        ⋅
 0.666667  1.0
permutation:
2-element Vector{Int64}:
 2
 1
```

`LinearAlgebra.bunchkaufman` – Function.

```
bunchkaufman(A, rook::Bool=false; check = true) -> S::BunchKaufman
```

Compute the Bunch-Kaufman [7] factorization of a symmetric or Hermitian matrix `A` as `P'*U*D*U'*P` or `P'*L*D*L'*P`, depending on which triangle is stored in `A`, and return a `BunchKaufman` object. Here, `U` and `L` are respectively upper and lower triangular, `P` is a permutation matrix, and `D` is block diagonal with 1-by-1 and 2-by-2 blocks. Note that if `A` is complex symmetric then `U'` and `L'` denote the unconjugated transposes, i.e. `transpose(U)` and `transpose(L)`.

Iterating the decomposition produces the components `S.D`, `S.U` or `S.L` as appropriate given `S.uplo`, and `S.p`.

If `rook` is `true`, rook pivoting is used. If `rook` is false, rook pivoting is not used.

When `check = true`, an error is thrown if the decomposition fails. When `check = false`, responsibility for checking the decomposition's validity (via `issuccess`) lies with the user.

The following functions are available for `BunchKaufman` objects: `size`, `\`, `inv`, `issymmetric`, `ishermitian`, `getindex`.

Examples

```
julia> A = Float64.([1 2; 2 3])
2×2 Matrix{Float64}:
 1.0  2.0
 2.0  3.0

julia> S = bunchkaufman(A) # A gets wrapped internally by Symmetric(A)
BunchKaufman{Float64, Matrix{Float64}, Vector{Int64}}
D factor:
```

[7] J R Bunch and L Kaufman, Some stable methods for calculating inertia and solving symmetric linear systems, Mathematics of Computation 31:137 (1977), 163-179. url.

```
2×2 Tridiagonal{Float64, Vector{Float64}}:
 -0.333333  0.0
  0.0       3.0
U factor:
2×2 UnitUpperTriangular{Float64, Matrix{Float64}}:
 1.0  0.666667
  ·   1.0
permutation:
2-element Vector{Int64}:
 1
 2

julia> d, u, p = S; # destructuring via iteration

julia> d == S.D && u == S.U && p == S.p
true

julia> S.U * S.D * S.U' ≈ S.P * A * S.P'
true

julia> S = bunchkaufman(Symmetric(A, :L))
BunchKaufman{Float64, Matrix{Float64}, Vector{Int64}}
D factor:
2×2 Tridiagonal{Float64, Vector{Float64}}:
 3.0   0.0
 0.0  -0.333333
L factor:
2×2 UnitLowerTriangular{Float64, Matrix{Float64}}:
 1.0        ·
 0.666667  1.0
permutation:
2-element Vector{Int64}:
 2
 1

julia> S.L * S.D * S.L' ≈ A[S.p, S.p]
true
```

`LinearAlgebra.bunchkaufman!` – Function.

```
bunchkaufman!(A, rook::Bool=false; check = true) -> BunchKaufman
```

`bunchkaufman!` is the same as `bunchkaufman`, but saves space by overwriting the input `A`, instead of creating a copy.

`LinearAlgebra.Eigen` – Type.

```
Eigen <: Factorization
```

Matrix factorization type of the eigenvalue/spectral decomposition of a square matrix `A`. This is the return type of `eigen`, the corresponding matrix factorization function.

If `F::Eigen` is the factorization object, the eigenvalues can be obtained via `F.values` and the eigenvectors as the columns of the matrix `F.vectors`. (The kth eigenvector can be obtained from the slice `F.vectors[:, k]`.)

Iterating the decomposition produces the components `F.values` and `F.vectors`.

Examples

```
julia> F = eigen([1.0 0.0 0.0; 0.0 3.0 0.0; 0.0 0.0 18.0])
Eigen{Float64, Float64, Matrix{Float64}, Vector{Float64}}
values:
3-element Vector{Float64}:
  1.0
  3.0
 18.0
vectors:
3×3 Matrix{Float64}:
 1.0  0.0  0.0
 0.0  1.0  0.0
 0.0  0.0  1.0

julia> F.values
3-element Vector{Float64}:
  1.0
  3.0
 18.0

julia> F.vectors
3×3 Matrix{Float64}:
 1.0  0.0  0.0
 0.0  1.0  0.0
 0.0  0.0  1.0

julia> vals, vecs = F; # destructuring via iteration

julia> vals == F.values && vecs == F.vectors
true
```

`LinearAlgebra.GeneralizedEigen` – Type.

```
GeneralizedEigen <: Factorization
```

Matrix factorization type of the generalized eigenvalue/spectral decomposition of `A` and `B`. This is the return type of `eigen`, the corresponding matrix factorization function, when called with two matrix arguments.

If `F::GeneralizedEigen` is the factorization object, the eigenvalues can be obtained via `F.values` and the eigenvectors as the columns of the matrix `F.vectors`. (The kth eigenvector can be obtained from the slice `F.vectors[:, k]`.)

Iterating the decomposition produces the components `F.values` and `F.vectors`.

Examples

```
julia> A = [1 0; 0 -1]
2×2 Matrix{Int64}:
 1   0
 0  -1
```

```
julia> B = [0 1; 1 0]
2×2 Matrix{Int64}:
 0  1
 1  0

julia> F = eigen(A, B)
GeneralizedEigen{ComplexF64, ComplexF64, Matrix{ComplexF64}, Vector{ComplexF64}}
values:
2-element Vector{ComplexF64}:
 0.0 - 1.0im
 0.0 + 1.0im
vectors:
2×2 Matrix{ComplexF64}:
  0.0+1.0im   0.0-1.0im
 -1.0+0.0im  -1.0-0.0im

julia> F.values
2-element Vector{ComplexF64}:
 0.0 - 1.0im
 0.0 + 1.0im

julia> F.vectors
2×2 Matrix{ComplexF64}:
  0.0+1.0im   0.0-1.0im
 -1.0+0.0im  -1.0-0.0im

julia> vals, vecs = F; # destructuring via iteration

julia> vals == F.values && vecs == F.vectors
true
```

`LinearAlgebra.eigvals` – Function.

```
eigvals(A; permute::Bool=true, scale::Bool=true, sortby) -> values
```

Return the eigenvalues of A.

For general non-symmetric matrices it is possible to specify how the matrix is balanced before the eigenvalue calculation. The `permute`, `scale`, and `sortby` keywords are the same as for `eigen`.

Examples

```
julia> diag_matrix = [1 0; 0 4]
2×2 Matrix{Int64}:
 1  0
 0  4

julia> eigvals(diag_matrix)
2-element Vector{Float64}:
 1.0
 4.0
```

For a scalar input, `eigvals` will return a scalar.

Examples

```
julia> eigvals(-2)
-2
```

```
eigvals(A, B) -> values
```

Compute the generalized eigenvalues of A and B.

Examples

```
julia> A = [1 0; 0 -1]
2×2 Matrix{Int64}:
 1   0
 0  -1

julia> B = [0 1; 1 0]
2×2 Matrix{Int64}:
 0  1
 1  0

julia> eigvals(A,B)
2-element Vector{ComplexF64}:
 0.0 - 1.0im
 0.0 + 1.0im
```

```
eigvals(A::Union{Hermitian, Symmetric}; alg::Algorithm = default_eigen_alg(A)) -> values
```

Return the eigenvalues of A.

`alg` specifies which algorithm and LAPACK method to use for eigenvalue decomposition:

- `alg = DivideAndConquer()`: Calls `LAPACK.syevd!`.
- `alg = QRIteration()`: Calls `LAPACK.syev!`.
- `alg = RobustRepresentations()` (default): Multiple relatively robust representations method, Calls `LAPACK.syevr!`.

See James W. Demmel et al, SIAM J. Sci. Comput. 30, 3, 1508 (2008) for a comparison of the accuracy and performance of different methods.

The default `alg` used may change in the future.

Julia 1.12

The `alg` keyword argument requires Julia 1.12 or later.

```
eigvals(A::Union{SymTridiagonal, Hermitian, Symmetric}, irange::UnitRange) -> values
```

Return the eigenvalues of A. It is possible to calculate only a subset of the eigenvalues by specifying a `UnitRange` `irange` covering indices of the sorted eigenvalues, e.g. the 2nd to 8th eigenvalues.

Examples

```
julia> A = SymTridiagonal([1.; 2.; 1.], [2.; 3.])
3×3 SymTridiagonal{Float64, Vector{Float64}}:
 1.0  2.0   ·
 2.0  2.0  3.0
  ·   3.0  1.0

julia> eigvals(A, 2:2)
1-element Vector{Float64}:
 0.9999999999999996

julia> eigvals(A)
3-element Vector{Float64}:
 -2.1400549446402604
  1.0000000000000002
  5.140054944640259
```

```
eigvals(A::Union{SymTridiagonal, Hermitian, Symmetric}, vl::Real, vu::Real) -> values
```

Return the eigenvalues of A. It is possible to calculate only a subset of the eigenvalues by specifying a pair `vl` and `vu` for the lower and upper boundaries of the eigenvalues.

Examples

```
julia> A = SymTridiagonal([1.; 2.; 1.], [2.; 3.])
3×3 SymTridiagonal{Float64, Vector{Float64}}:
 1.0  2.0   ·
 2.0  2.0  3.0
  ·   3.0  1.0

julia> eigvals(A, -1, 2)
1-element Vector{Float64}:
 1.0000000000000009

julia> eigvals(A)
3-element Vector{Float64}:
 -2.1400549446402604
  1.0000000000000002
  5.140054944640259
```

`LinearAlgebra.eigvals!` – Function.

```
eigvals!(A; permute::Bool=true, scale::Bool=true, sortby) -> values
```

Same as `eigvals`, but saves space by overwriting the input A, instead of creating a copy. The `permute`, `scale`, and `sortby` keywords are the same as for `eigen`.

Note

The input matrix A will not contain its eigenvalues after `eigvals!` is called on it - A is used as a workspace.

Examples

```
julia> A = [1. 2.; 3. 4.]
2×2 Matrix{Float64}:
 1.0  2.0
 3.0  4.0

julia> eigvals!(A)
2-element Vector{Float64}:
 -0.3722813232690143
  5.372281323269014

julia> A
2×2 Matrix{Float64}:
 -0.372281  -1.0
  0.0        5.37228
```

```
eigvals!(A, B; sortby) -> values
```

Same as `eigvals`, but saves space by overwriting the input A (and B), instead of creating copies.

Note

The input matrices A and B will not contain their eigenvalues after `eigvals!` is called. They are used as workspaces.

Examples

```
julia> A = [1. 0.; 0. -1.]
2×2 Matrix{Float64}:
 1.0   0.0
 0.0  -1.0

julia> B = [0. 1.; 1. 0.]
2×2 Matrix{Float64}:
 0.0  1.0
 1.0  0.0

julia> eigvals!(A, B)
2-element Vector{ComplexF64}:
 0.0 - 1.0im
 0.0 + 1.0im

julia> A
2×2 Matrix{Float64}:
 -0.0  -1.0
  1.0  -0.0

julia> B
2×2 Matrix{Float64}:
 1.0  0.0
 0.0  1.0
```

```
eigvals!(A::Union{SymTridiagonal, Hermitian, Symmetric}, irange::UnitRange) -> values
```

Same as eigvals, but saves space by overwriting the input A, instead of creating a copy. irange is a range of eigenvalue *indices* to search for - for instance, the 2nd to 8th eigenvalues.

```
eigvals!(A::Union{SymTridiagonal, Hermitian, Symmetric}, vl::Real, vu::Real) -> values
```

Same as eigvals, but saves space by overwriting the input A, instead of creating a copy. vl is the lower bound of the interval to search for eigenvalues, and vu is the upper bound.

LinearAlgebra.eigmax – Function.

```
eigmax(A; permute::Bool=true, scale::Bool=true)
```

Return the largest eigenvalue of A. The option permute=true permutes the matrix to become closer to upper triangular, and scale=true scales the matrix by its diagonal elements to make rows and columns more equal in norm. Note that if the eigenvalues of A are complex, this method will fail, since complex numbers cannot be sorted.

Examples

```
julia> A = [0 im; -im 0]
2×2 Matrix{Complex{Int64}}:
 0+0im  0+1im
 0-1im  0+0im

julia> eigmax(A)
1.0

julia> A = [0 im; -1 0]
2×2 Matrix{Complex{Int64}}:
  0+0im  0+1im
 -1+0im  0+0im

julia> eigmax(A)
ERROR: DomainError with Complex{Int64}[0+0im 0+1im; -1+0im 0+0im]:
`A` cannot have complex eigenvalues.
Stacktrace:
[...]
```

LinearAlgebra.eigmin – Function.

```
eigmin(A; permute::Bool=true, scale::Bool=true)
```

Return the smallest eigenvalue of A. The option permute=true permutes the matrix to become closer to upper triangular, and scale=true scales the matrix by its diagonal elements to make rows and columns more equal in norm. Note that if the eigenvalues of A are complex, this method will fail, since complex numbers cannot be sorted.

Examples

```
julia> A = [0 im; -im 0]
2×2 Matrix{Complex{Int64}}:
 0+0im  0+1im
 0-1im  0+0im
```

```
julia> eigmin(A)
-1.0

julia> A = [0 im; -1 0]
2×2 Matrix{Complex{Int64}}:
  0+0im  0+1im
 -1+0im  0+0im

julia> eigmin(A)
ERROR: DomainError with Complex{Int64}[0+0im 0+1im; -1+0im 0+0im]:
`A` cannot have complex eigenvalues.
Stacktrace:
[...]
```

`LinearAlgebra.eigvecs` – Function.

```
eigvecs(A::Union{Symmetric, Hermitian, SymTridiagonal}[, eigvals])::Matrix
```

Return a matrix `M` whose columns are the eigenvectors of `A`. (The kth eigenvector can be obtained from the slice `M[:, k]`.)

If the optional vector of eigenvalues `eigvals` is specified, `eigvecs` returns the specific corresponding eigenvectors.

Examples

```
julia> A = SymTridiagonal([1.; 2.; 1.], [2.; 3.])
3×3 SymTridiagonal{Float64, Vector{Float64}}:
 1.0  2.0   ·
 2.0  2.0  3.0
  ·   3.0  1.0

julia> eigvals(A)
3-element Vector{Float64}:
 -2.1400549446402604
  1.0000000000000002
  5.140054944640259

julia> eigvecs(A)
3×3 Matrix{Float64}:
  0.418304  -0.83205      0.364299
 -0.656749  -7.39009e-16  0.754109
  0.627457   0.5547       0.546448

julia> eigvecs(A, [1.])
3×1 Matrix{Float64}:
  0.8320502943378438
  4.263514128092366e-17
 -0.5547001962252291
```

```
eigvecs(A; permute::Bool=true, scale::Bool=true, `sortby`) -> Matrix
```

Return a matrix `M` whose columns are the eigenvectors of `A`. (The kth eigenvector can be obtained from the slice `M[:, k]`.) The `permute`, `scale`, and `sortby` keywords are the same as for `eigen`.

Examples

```
julia> eigvecs([1.0 0.0 0.0; 0.0 3.0 0.0; 0.0 0.0 18.0])
3×3 Matrix{Float64}:
 1.0  0.0  0.0
 0.0  1.0  0.0
 0.0  0.0  1.0
```

```
eigvecs(A, B) -> Matrix
```

Return a matrix `M` whose columns are the generalized eigenvectors of `A` and `B`. (The kth eigenvector can be obtained from the slice `M[:, k]`.)

Examples

```
julia> A = [1 0; 0 -1]
2×2 Matrix{Int64}:
 1   0
 0  -1

julia> B = [0 1; 1 0]
2×2 Matrix{Int64}:
 0  1
 1  0

julia> eigvecs(A, B)
2×2 Matrix{ComplexF64}:
  0.0+1.0im   0.0-1.0im
 -1.0+0.0im  -1.0-0.0im
```

`LinearAlgebra.eigen` – Function.

```
eigen(A; permute::Bool=true, scale::Bool=true, sortby) -> Eigen
```

Compute the eigenvalue decomposition of `A`, returning an `Eigen` factorization object `F` which contains the eigenvalues in `F.values` and the normalized eigenvectors in the columns of the matrix `F.vectors`. This corresponds to solving an eigenvalue problem of the form `Ax = λx`, where `A` is a matrix, `x` is an eigenvector, and `λ` is an eigenvalue. (The kth eigenvector can be obtained from the slice `F.vectors[:, k]`.)

Iterating the decomposition produces the components `F.values` and `F.vectors`.

The following functions are available for `Eigen` objects: `inv`, `det`, and `isposdef`.

For general nonsymmetric matrices it is possible to specify how the matrix is balanced before the eigenvector calculation. The option `permute=true` permutes the matrix to become closer to upper triangular, and `scale=true` scales the matrix by its diagonal elements to make rows and columns more equal in norm. The default is `true` for both options.

By default, the eigenvalues and vectors are sorted lexicographically by `(real(λ),imag(λ))`. A different comparison function `by(λ)` can be passed to `sortby`, or you can pass `sortby=nothing` to leave the

eigenvalues in an arbitrary order. Some special matrix types (e.g. `Diagonal` or `SymTridiagonal`) may implement their own sorting convention and not accept a `sortby` keyword.

Examples

```
julia> F = eigen([1.0 0.0 0.0; 0.0 3.0 0.0; 0.0 0.0 18.0])
Eigen{Float64, Float64, Matrix{Float64}, Vector{Float64}}
values:
3-element Vector{Float64}:
  1.0
  3.0
 18.0
vectors:
3×3 Matrix{Float64}:
 1.0  0.0  0.0
 0.0  1.0  0.0
 0.0  0.0  1.0

julia> F.values
3-element Vector{Float64}:
  1.0
  3.0
 18.0

julia> F.vectors
3×3 Matrix{Float64}:
 1.0  0.0  0.0
 0.0  1.0  0.0
 0.0  0.0  1.0

julia> vals, vecs = F; # destructuring via iteration

julia> vals == F.values && vecs == F.vectors
true
```

```
eigen(A, B; sortby) -> GeneralizedEigen
```

Compute the generalized eigenvalue decomposition of `A` and `B`, returning a `GeneralizedEigen` factorization object `F` which contains the generalized eigenvalues in `F.values` and the generalized eigenvectors in the columns of the matrix `F.vectors`. This corresponds to solving a generalized eigenvalue problem of the form `Ax = λBx`, where `A, B` are matrices, `x` is an eigenvector, and `λ` is an eigenvalue. (The kth generalized eigenvector can be obtained from the slice `F.vectors[:, k]`.)

Iterating the decomposition produces the components `F.values` and `F.vectors`.

By default, the eigenvalues and vectors are sorted lexicographically by `(real(λ),imag(λ))`. A different comparison function `by(λ)` can be passed to `sortby`, or you can pass `sortby=nothing` to leave the eigenvalues in an arbitrary order.

Examples

```
julia> A = [1 0; 0 -1]
2×2 Matrix{Int64}:
 1   0
```

```
 0  -1

julia> B = [0 1; 1 0]
2×2 Matrix{Int64}:
 0  1
 1  0

julia> F = eigen(A, B);

julia> F.values
2-element Vector{ComplexF64}:
 0.0 - 1.0im
 0.0 + 1.0im

julia> F.vectors
2×2 Matrix{ComplexF64}:
  0.0+1.0im   0.0-1.0im
 -1.0+0.0im  -1.0-0.0im

julia> vals, vecs = F; # destructuring via iteration

julia> vals == F.values && vecs == F.vectors
true
```

```
eigen(A::Union{Hermitian, Symmetric}; alg::LinearAlgebra.Algorithm =
↪  LinearAlgebra.default_eigen_alg(A)) -> Eigen
```

Compute the eigenvalue decomposition of `A`, returning an `Eigen` factorization object `F` which contains the eigenvalues in `F.values` and the orthonormal eigenvectors in the columns of the matrix `F.vectors`. (The kth eigenvector can be obtained from the slice `F.vectors[:, k]`.)

Iterating the decomposition produces the components `F.values` and `F.vectors`.

`alg` specifies which algorithm and LAPACK method to use for eigenvalue decomposition:

- `alg = DivideAndConquer()`: Calls `LAPACK.syevd!`.
- `alg = QRIteration()`: Calls `LAPACK.syev!`.
- `alg = RobustRepresentations()` (default): Multiple relatively robust representations method, Calls `LAPACK.syevr!`.

See James W. Demmel et al, SIAM J. Sci. Comput. 30, 3, 1508 (2008) for a comparison of the accuracy and performance of different algorithms.

The default `alg` used may change in the future.

Julia 1.12

The `alg` keyword argument requires Julia 1.12 or later.

The following functions are available for `Eigen` objects: `inv`, `det`, and `isposdef`.

```
eigen(A::Union{SymTridiagonal, Hermitian, Symmetric}, irange::UnitRange) -> Eigen
```

Compute the eigenvalue decomposition of A, returning an Eigen factorization object F which contains the eigenvalues in F.values and the orthonormal eigenvectors in the columns of the matrix F.vectors. (The kth eigenvector can be obtained from the slice F.vectors[:, k].)

Iterating the decomposition produces the components F.values and F.vectors.

The following functions are available for Eigen objects: inv, det, and isposdef.

The UnitRange irange specifies indices of the sorted eigenvalues to search for.

Note

If irange is not 1:n, where n is the dimension of A, then the returned factorization will be a *truncated* factorization.

```
eigen(A::Union{SymTridiagonal, Hermitian, Symmetric}, vl::Real, vu::Real) -> Eigen
```

Compute the eigenvalue decomposition of A, returning an Eigen factorization object F which contains the eigenvalues in F.values and the orthonormal eigenvectors in the columns of the matrix F.vectors. (The kth eigenvector can be obtained from the slice F.vectors[:, k].)

Iterating the decomposition produces the components F.values and F.vectors.

The following functions are available for Eigen objects: inv, det, and isposdef.

vl is the lower bound of the window of eigenvalues to search for, and vu is the upper bound.

Note

If [vl, vu] does not contain all eigenvalues of A, then the returned factorization will be a *truncated* factorization.

LinearAlgebra.eigen! – Function.

```
eigen!(A; permute, scale, sortby)
eigen!(A, B; sortby)
```

Same as eigen, but saves space by overwriting the input A (and B), instead of creating a copy.

LinearAlgebra.Hessenberg – Type.

```
Hessenberg <: Factorization
```

A Hessenberg object represents the Hessenberg factorization QHQ' of a square matrix, or a shift Q(H+μI)Q' thereof, which is produced by the hessenberg function.

LinearAlgebra.hessenberg – Function.

```
hessenberg(A) -> Hessenberg
```

Compute the Hessenberg decomposition of A and return a Hessenberg object. If F is the factorization object, the unitary matrix can be accessed with F.Q (of type LinearAlgebra.HessenbergQ) and the Hessenberg matrix with F.H (of type UpperHessenberg), either of which may be converted to a regular matrix with Matrix(F.H) or Matrix(F.Q).

If A is Hermitian or real-Symmetric, then the Hessenberg decomposition produces a real-symmetric tridiagonal matrix and F.H is of type SymTridiagonal.

Note that the shifted factorization A+μI = Q (H+μI) Q' can be constructed efficiently by F + μ*I using the UniformScaling object I, which creates a new Hessenberg object with shared storage and a modified shift. The shift of a given F is obtained by F.μ. This is useful because multiple shifted solves (F + μ*I) \ b (for different μ and/or b) can be performed efficiently once F is created.

Iterating the decomposition produces the factors F.Q, F.H, F.μ.

Examples

```
julia> A = [4. 9. 7.; 4. 4. 1.; 4. 3. 2.]
3×3 Matrix{Float64}:
 4.0  9.0  7.0
 4.0  4.0  1.0
 4.0  3.0  2.0

julia> F = hessenberg(A)
Hessenberg{Float64, UpperHessenberg{Float64, Matrix{Float64}}, Matrix{Float64},
↪  Vector{Float64}, Bool}
Q factor: 3×3 LinearAlgebra.HessenbergQ{Float64, Matrix{Float64}, Vector{Float64}, false}
H factor:
3×3 UpperHessenberg{Float64, Matrix{Float64}}:
  4.0      -11.3137       -1.41421
 -5.65685    5.0           2.0
   ·        -8.88178e-16   1.0

julia> F.Q * F.H * F.Q'
3×3 Matrix{Float64}:
 4.0  9.0  7.0
 4.0  4.0  1.0
 4.0  3.0  2.0

julia> q, h = F; # destructuring via iteration

julia> q == F.Q && h == F.H
true
```

LinearAlgebra.hessenberg! – Function.

```
hessenberg!(A) -> Hessenberg
```

hessenberg! is the same as hessenberg, but saves space by overwriting the input A, instead of creating a copy.

LinearAlgebra.Schur – Type.

```
Schur <: Factorization
```

Matrix factorization type of the Schur factorization of a matrix A. This is the return type of schur(_), the corresponding matrix factorization function.

If F::Schur is the factorization object, the (quasi) triangular Schur factor can be obtained via either F.Schur or F.T and the orthogonal/unitary Schur vectors via F.vectors or F.Z such that A = F.vectors * F.Schur * F.vectors'. The eigenvalues of A can be obtained with F.values.

Iterating the decomposition produces the components `F.T`, `F.Z`, and `F.values`.

Examples

```
julia> A = [5. 7.; -2. -4.]
2×2 Matrix{Float64}:
  5.0   7.0
 -2.0  -4.0

julia> F = schur(A)
Schur{Float64, Matrix{Float64}, Vector{Float64}}
T factor:
2×2 Matrix{Float64}:
 3.0   9.0
 0.0  -2.0
Z factor:
2×2 Matrix{Float64}:
  0.961524  0.274721
 -0.274721  0.961524
eigenvalues:
2-element Vector{Float64}:
  3.0
 -2.0

julia> F.vectors * F.Schur * F.vectors'
2×2 Matrix{Float64}:
  5.0   7.0
 -2.0  -4.0

julia> t, z, vals = F; # destructuring via iteration

julia> t == F.T && z == F.Z && vals == F.values
true
```

`LinearAlgebra.GeneralizedSchur` – Type.

```
GeneralizedSchur <: Factorization
```

Matrix factorization type of the generalized Schur factorization of two matrices `A` and `B`. This is the return type of `schur(_, _)`, the corresponding matrix factorization function.

If `F::GeneralizedSchur` is the factorization object, the (quasi) triangular Schur factors can be obtained via `F.S` and `F.T`, the left unitary/orthogonal Schur vectors via `F.left` or `F.Q`, and the right unitary/orthogonal Schur vectors can be obtained with `F.right` or `F.Z` such that `A=F.left*F.S*F.right'` and `B=F.left*F.T*F.right'`. The generalized eigenvalues of `A` and `B` can be obtained with `F.α./F.β`.

Iterating the decomposition produces the components `F.S`, `F.T`, `F.Q`, `F.Z`, `F.α`, and `F.β`.

`LinearAlgebra.schur` – Function.

```
schur(A) -> F::Schur
```

Computes the Schur factorization of the matrix `A`. The (quasi) triangular Schur factor can be obtained from the `Schur` object `F` with either `F.Schur` or `F.T` and the orthogonal/unitary Schur vectors can be obtained with `F.vectors` or `F.Z` such that `A = F.vectors * F.Schur * F.vectors'`. The eigenvalues of `A` can be obtained with `F.values`.

For real A, the Schur factorization is "quasitriangular", which means that it is upper-triangular except with 2×2 diagonal blocks for any conjugate pair of complex eigenvalues; this allows the factorization to be purely real even when there are complex eigenvalues. To obtain the (complex) purely upper-triangular Schur factorization from a real quasitriangular factorization, you can use `Schur{Complex}(schur(A))`.

Iterating the decomposition produces the components `F.T`, `F.Z`, and `F.values`.

Examples

```
julia> A = [5. 7.; -2. -4.]
2×2 Matrix{Float64}:
  5.0   7.0
 -2.0  -4.0

julia> F = schur(A)
Schur{Float64, Matrix{Float64}, Vector{Float64}}
T factor:
2×2 Matrix{Float64}:
 3.0   9.0
 0.0  -2.0
Z factor:
2×2 Matrix{Float64}:
  0.961524  0.274721
 -0.274721  0.961524
eigenvalues:
2-element Vector{Float64}:
  3.0
 -2.0

julia> F.vectors * F.Schur * F.vectors'
2×2 Matrix{Float64}:
  5.0   7.0
 -2.0  -4.0

julia> t, z, vals = F; # destructuring via iteration

julia> t == F.T && z == F.Z && vals == F.values
true
```

```
schur(A, B) -> F::GeneralizedSchur
```

Computes the Generalized Schur (or QZ) factorization of the matrices A and B. The (quasi) triangular Schur factors can be obtained from the `Schur` object F with `F.S` and `F.T`, the left unitary/orthogonal Schur vectors can be obtained with `F.left` or `F.Q` and the right unitary/orthogonal Schur vectors can be obtained with `F.right` or `F.Z` such that `A=F.left*F.S*F.right'` and `B=F.left*F.T*F.right'`. The generalized eigenvalues of A and B can be obtained with `F.α./F.β`.

Iterating the decomposition produces the components `F.S`, `F.T`, `F.Q`, `F.Z`, `F.α`, and `F.β`.

`LinearAlgebra.schur!` – Function.

```
schur!(A) -> F::Schur
```

Same as `schur` but uses the input argument A as workspace.

Examples

```
julia> A = [5. 7.; -2. -4.]
2×2 Matrix{Float64}:
  5.0   7.0
 -2.0  -4.0

julia> F = schur!(A)
Schur{Float64, Matrix{Float64}, Vector{Float64}}
T factor:
2×2 Matrix{Float64}:
 3.0   9.0
 0.0  -2.0
Z factor:
2×2 Matrix{Float64}:
  0.961524  0.274721
 -0.274721  0.961524
eigenvalues:
2-element Vector{Float64}:
  3.0
 -2.0

julia> A
2×2 Matrix{Float64}:
 3.0   9.0
 0.0  -2.0
```

```
schur!(A::StridedMatrix, B::StridedMatrix) -> F::GeneralizedSchur
```

Same as schur but uses the input matrices A and B as workspace.

`LinearAlgebra.ordschur` – Function.

```
ordschur(F::Schur, select::Union{Vector{Bool},BitVector}) -> F::Schur
```

Reorders the Schur factorization F of a matrix `A = Z*T*Z'` according to the logical array `select` returning the reordered factorization F object. The selected eigenvalues appear in the leading diagonal of `F.Schur` and the corresponding leading columns of `F.vectors` form an orthogonal/unitary basis of the corresponding right invariant subspace. In the real case, a complex conjugate pair of eigenvalues must be either both included or both excluded via `select`.

```
ordschur(F::GeneralizedSchur, select::Union{Vector{Bool},BitVector}) -> F::GeneralizedSchur
```

Reorders the Generalized Schur factorization F of a matrix pair `(A, B) = (Q*S*Z', Q*T*Z')` according to the logical array `select` and returns a GeneralizedSchur object F. The selected eigenvalues appear in the leading diagonal of both `F.S` and `F.T`, and the left and right orthogonal/unitary Schur vectors are also reordered such that `(A, B) = F.Q*(F.S, F.T)*F.Z'` still holds and the generalized eigenvalues of A and B can still be obtained with `F.α./F.β`.

`LinearAlgebra.ordschur!` – Function.

```
ordschur!(F::Schur, select::Union{Vector{Bool},BitVector}) -> F::Schur
```

Same as ordschur but overwrites the factorization F.

```
ordschur!(F::GeneralizedSchur, select::Union{Vector{Bool},BitVector}) -> F::GeneralizedSchur
```

Same as `ordschur` but overwrites the factorization `F`.

`LinearAlgebra.SVD` – Type.

```
SVD <: Factorization
```

Matrix factorization type of the singular value decomposition (SVD) of a matrix `A`. This is the return type of `svd(_)`, the corresponding matrix factorization function.

If `F::SVD` is the factorization object, `U`, `S`, `V` and `Vt` can be obtained via `F.U`, `F.S`, `F.V` and `F.Vt`, such that `A = U * Diagonal(S) * Vt`. The singular values in `S` are sorted in descending order.

Iterating the decomposition produces the components `U`, `S`, and `V`.

Examples

```
julia> A = [1. 0. 0. 0. 2.; 0. 0. 3. 0. 0.; 0. 0. 0. 0. 0.; 0. 2. 0. 0. 0.]
4×5 Matrix{Float64}:
 1.0  0.0  0.0  0.0  2.0
 0.0  0.0  3.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0
 0.0  2.0  0.0  0.0  0.0

julia> F = svd(A)
SVD{Float64, Float64, Matrix{Float64}, Vector{Float64}}
U factor:
4×4 Matrix{Float64}:
 0.0  1.0   0.0  0.0
 1.0  0.0   0.0  0.0
 0.0  0.0   0.0  1.0
 0.0  0.0  -1.0  0.0
singular values:
4-element Vector{Float64}:
 3.0
 2.23606797749979
 2.0
 0.0
Vt factor:
4×5 Matrix{Float64}:
 -0.0        0.0  1.0  -0.0  0.0
  0.447214   0.0  0.0   0.0  0.894427
  0.0       -1.0  0.0   0.0  0.0
  0.0        0.0  0.0   1.0  0.0

julia> F.U * Diagonal(F.S) * F.Vt
4×5 Matrix{Float64}:
 1.0  0.0  0.0  0.0  2.0
 0.0  0.0  3.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0
 0.0  2.0  0.0  0.0  0.0

julia> u, s, v = F; # destructuring via iteration
```

```
julia> u == F.U && s == F.S && v == F.V
true
```

`LinearAlgebra.GeneralizedSVD` – Type.

```
GeneralizedSVD <: Factorization
```

Matrix factorization type of the generalized singular value decomposition (SVD) of two matrices `A` and `B`, such that `A = F.U*F.D1*F.R0*F.Q'` and `B = F.V*F.D2*F.R0*F.Q'`. This is the return type of `svd(_, _)`, the corresponding matrix factorization function.

For an M-by-N matrix `A` and P-by-N matrix `B`,

- `U` is a M-by-M orthogonal matrix,
- `V` is a P-by-P orthogonal matrix,
- `Q` is a N-by-N orthogonal matrix,
- `D1` is a M-by-(K+L) diagonal matrix with 1s in the first K entries,
- `D2` is a P-by-(K+L) matrix whose top right L-by-L block is diagonal,
- `R0` is a (K+L)-by-N matrix whose rightmost (K+L)-by-(K+L) block is nonsingular upper block triangular,

K+L is the effective numerical rank of the matrix `[A; B]`.

Iterating the decomposition produces the components `U`, `V`, `Q`, `D1`, `D2`, and `R0`.

The entries of `F.D1` and `F.D2` are related, as explained in the LAPACK documentation for the generalized SVD and the xGGSVD3 routine which is called underneath (in LAPACK 3.6.0 and newer).

Examples

```
julia> A = [1. 0.; 0. -1.]
2×2 Matrix{Float64}:
 1.0   0.0
 0.0  -1.0

julia> B = [0. 1.; 1. 0.]
2×2 Matrix{Float64}:
 0.0  1.0
 1.0  0.0

julia> F = svd(A, B)
GeneralizedSVD{Float64, Matrix{Float64}, Float64, Vector{Float64}}
U factor:
2×2 Matrix{Float64}:
 1.0  0.0
 0.0  1.0
V factor:
2×2 Matrix{Float64}:
 -0.0  -1.0
  1.0   0.0
Q factor:
2×2 Matrix{Float64}:
```

```
 1.0  0.0
 0.0  1.0
D1 factor:
2×2 Matrix{Float64}:
 0.707107  0.0
 0.0       0.707107
D2 factor:
2×2 Matrix{Float64}:
 0.707107  0.0
 0.0       0.707107
R0 factor:
2×2 Matrix{Float64}:
 1.41421   0.0
 0.0      -1.41421

julia> F.U*F.D1*F.R0*F.Q'
2×2 Matrix{Float64}:
 1.0   0.0
 0.0  -1.0

julia> F.V*F.D2*F.R0*F.Q'
2×2 Matrix{Float64}:
 -0.0  1.0
  1.0  0.0
```

`LinearAlgebra.svd` – Function.

```
svd(A; full::Bool = false, alg::Algorithm = default_svd_alg(A), atol::Real=0, rtol::Real=0) ->
↪  SVD
```

Compute the singular value decomposition (SVD) of `A` and return an `SVD` object.

`U`, `S`, `V` and `Vt` can be obtained from the factorization `F` with `F.U`, `F.S`, `F.V` and `F.Vt`, such that `A = U * Diagonal(S) * Vt`. The algorithm produces `Vt` and hence `Vt` is more efficient to extract than `V`. The singular values in `S` are sorted in descending order.

Iterating the decomposition produces the components `U`, `S`, and `V`.

If `full = false` (default), a "thin" SVD is returned. For an $M \times N$ matrix `A`, in the full factorization `U` is $M \times M$ and `V` is $N \times N$, while in the thin factorization `U` is $M \times K$ and `V` is $N \times K$, where $K = \min(M, N)$ is the number of singular values.

`alg` specifies which algorithm and LAPACK method to use for SVD:

- `alg = LinearAlgebra.DivideAndConquer()` (default): Calls `LAPACK.gesdd!`.
- `alg = LinearAlgebra.QRIteration()`: Calls `LAPACK.gesvd!` (typically slower but more accurate).

The `atol` and `rtol` parameters specify optional tolerances to truncate the SVD, dropping (setting to zero) singular values less than `max(atol, rtol*σ₁)` where `σ₁` is the largest singular value of `A`.

Julia 1.3

The `alg` keyword argument requires Julia 1.3 or later.

Julia 1.13

The `atol` and `rtol` arguments require Julia 1.13 or later.

Examples

```
julia> A = rand(4,3);

julia> F = svd(A); # Store the Factorization Object

julia> A ≈ F.U * Diagonal(F.S) * F.Vt
true

julia> U, S, V = F; # destructuring via iteration

julia> A ≈ U * Diagonal(S) * V'
true

julia> Uonly, = svd(A); # Store U only

julia> Uonly == U
true
```

```
svd(A, B) -> GeneralizedSVD
```

Compute the generalized SVD of `A` and `B`, returning a `GeneralizedSVD` factorization object `F` such that `[A;B] = [F.U * F.D1; F.V * F.D2] * F.R0 * F.Q'`

- `U` is a M-by-M orthogonal matrix,
- `V` is a P-by-P orthogonal matrix,
- `Q` is a N-by-N orthogonal matrix,
- `D1` is a M-by-(K+L) diagonal matrix with 1s in the first K entries,
- `D2` is a P-by-(K+L) matrix whose top right L-by-L block is diagonal,
- `R0` is a (K+L)-by-N matrix whose rightmost (K+L)-by-(K+L) block is nonsingular upper block triangular,

K+L is the effective numerical rank of the matrix `[A; B]`.

Iterating the decomposition produces the components `U`, `V`, `Q`, `D1`, `D2`, and `R0`.

The generalized SVD is used in applications such as when one wants to compare how much belongs to `A` vs. how much belongs to `B`, as in human vs yeast genome, or signal vs noise, or between clusters vs within clusters. (See Edelman and Wang for discussion: https://arxiv.org/abs/1901.00485)

It decomposes `[A; B]` into `[UC; VS]H`, where `[UC; VS]` is a natural orthogonal basis for the column space of `[A; B]`, and `H = RQ'` is a natural non-orthogonal basis for the rowspace of `[A;B]`, where the top rows are most closely attributed to the `A` matrix, and the bottom to the `B` matrix. The multi-cosine/sine matrices `C` and `S` provide a multi-measure of how much `A` vs how much `B`, and `U` and `V` provide directions in which these are measured.

Examples

```
julia> A = randn(3,2); B=randn(4,2);

julia> F = svd(A, B);

julia> U,V,Q,C,S,R = F;

julia> H = R*Q';

julia> [A; B] ≈ [U*C; V*S]*H
true

julia> [A; B] ≈ [F.U*F.D1; F.V*F.D2]*F.R0*F.Q'
true

julia> Uonly, = svd(A,B);

julia> U == Uonly
true
```

`LinearAlgebra.svd!` – Function.

```
svd!(A; full::Bool = false, alg::Algorithm = default_svd_alg(A), atol::Real=0, rtol::Real=0)
↪  -> SVD
```

`svd!` is the same as `svd`, but saves space by overwriting the input `A`, instead of creating a copy. See documentation of `svd` for details.

```
svd!(A, B) -> GeneralizedSVD
```

`svd!` is the same as `svd`, but modifies the arguments `A` and `B` in-place, instead of making copies. See documentation of `svd` for details.

`LinearAlgebra.svdvals` – Function.

```
svdvals(A)
```

Return the singular values of `A` in descending order.

Examples

```
julia> A = [1. 0. 0. 0. 2.; 0. 0. 3. 0. 0.; 0. 0. 0. 0. 0.; 0. 2. 0. 0. 0.]
4×5 Matrix{Float64}:
 1.0  0.0  0.0  0.0  2.0
 0.0  0.0  3.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0
 0.0  2.0  0.0  0.0  0.0

julia> svdvals(A)
4-element Vector{Float64}:
 3.0
 2.23606797749979
 2.0
 0.0
```

```
svdvals(A, B)
```

Return the generalized singular values from the generalized singular value decomposition of A and B. See also svd.

Examples

```
julia> A = [1. 0.; 0. -1.]
2×2 Matrix{Float64}:
 1.0   0.0
 0.0  -1.0

julia> B = [0. 1.; 1. 0.]
2×2 Matrix{Float64}:
 0.0  1.0
 1.0  0.0

julia> svdvals(A, B)
2-element Vector{Float64}:
 1.0
 1.0
```

LinearAlgebra.svdvals! – Function.

```
svdvals!(A)
```

Return the singular values of A, saving space by overwriting the input. See also svdvals and svd.

```
svdvals!(A, B)
```

Return the generalized singular values from the generalized singular value decomposition of A and B, saving space by overwriting A and B. See also svd and svdvals.

LinearAlgebra.Givens – Type.

```
LinearAlgebra.Givens(i1,i2,c,s) -> G
```

A Givens rotation linear operator. The fields c and s represent the cosine and sine of the rotation angle, respectively. The Givens type supports left multiplication G*A and conjugated transpose right multiplication A*G'. The type doesn't have a size and can therefore be multiplied with matrices of arbitrary size as long as i2<=size(A,2) for G*A or i2<=size(A,1) for A*G'.

See also givens.

LinearAlgebra.givens – Function.

```
givens(f::T, g::T, i1::Integer, i2::Integer) where {T} -> (G::Givens, r::T)
```

Computes the Givens rotation G and scalar r such that for any vector x where

```
x[i1] = f
x[i2] = g
```

the result of the multiplication

```
y = G*x
```

has the property that

```
y[i1] = r
y[i2] = 0
```

See also LinearAlgebra.Givens.

```
givens(A::AbstractArray, i1::Integer, i2::Integer, j::Integer) -> (G::Givens, r)
```

Computes the Givens rotation G and scalar r such that the result of the multiplication

```
B = G*A
```

has the property that

```
B[i1,j] = r
B[i2,j] = 0
```

See also LinearAlgebra.Givens.

```
givens(x::AbstractVector, i1::Integer, i2::Integer) -> (G::Givens, r)
```

Computes the Givens rotation G and scalar r such that the result of the multiplication

```
B = G*x
```

has the property that

```
B[i1] = r
B[i2] = 0
```

See also LinearAlgebra.Givens.

LinearAlgebra.triu – Function.

```
triu(M, k::Integer = 0)
```

Return the upper triangle of M starting from the kth superdiagonal.

Examples

```
julia> a = fill(1.0, (4,4))
4×4 Matrix{Float64}:
 1.0  1.0  1.0  1.0
 1.0  1.0  1.0  1.0
 1.0  1.0  1.0  1.0
 1.0  1.0  1.0  1.0

julia> triu(a,3)
4×4 Matrix{Float64}:
```

```
 0.0  0.0  0.0  1.0
 0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0

julia> triu(a,-3)
4×4 Matrix{Float64}:
 1.0  1.0  1.0  1.0
 1.0  1.0  1.0  1.0
 1.0  1.0  1.0  1.0
 1.0  1.0  1.0  1.0
```

`LinearAlgebra.triu!` – Function.

```
triu!(M)
```

Upper triangle of a matrix, overwriting `M` in the process. See also `triu`.

```
triu!(M, k::Integer)
```

Return the upper triangle of `M` starting from the kth superdiagonal, overwriting `M` in the process.

Examples

```
julia> M = [1 2 3 4 5; 1 2 3 4 5; 1 2 3 4 5; 1 2 3 4 5; 1 2 3 4 5]
5×5 Matrix{Int64}:
 1  2  3  4  5
 1  2  3  4  5
 1  2  3  4  5
 1  2  3  4  5
 1  2  3  4  5

julia> triu!(M, 1)
5×5 Matrix{Int64}:
 0  2  3  4  5
 0  0  3  4  5
 0  0  0  4  5
 0  0  0  0  5
 0  0  0  0  0
```

`LinearAlgebra.tril` – Function.

```
tril(M, k::Integer = 0)
```

Return the lower triangle of `M` starting from the kth superdiagonal.

Examples

```
julia> a = fill(1.0, (4,4))
4×4 Matrix{Float64}:
 1.0  1.0  1.0  1.0
 1.0  1.0  1.0  1.0
 1.0  1.0  1.0  1.0
 1.0  1.0  1.0  1.0
```

```
julia> tril(a,3)
4×4 Matrix{Float64}:
 1.0  1.0  1.0  1.0
 1.0  1.0  1.0  1.0
 1.0  1.0  1.0  1.0
 1.0  1.0  1.0  1.0

julia> tril(a,-3)
4×4 Matrix{Float64}:
 0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0
 1.0  0.0  0.0  0.0
```

`LinearAlgebra.tril!` – Function.

```
tril!(M)
```

Lower triangle of a matrix, overwriting `M` in the process. See also `tril`.

```
tril!(M, k::Integer)
```

Return the lower triangle of `M` starting from the kth superdiagonal, overwriting `M` in the process.

Examples

```
julia> M = [1 2 3 4 5; 1 2 3 4 5; 1 2 3 4 5; 1 2 3 4 5; 1 2 3 4 5]
5×5 Matrix{Int64}:
 1  2  3  4  5
 1  2  3  4  5
 1  2  3  4  5
 1  2  3  4  5
 1  2  3  4  5

julia> tril!(M, 2)
5×5 Matrix{Int64}:
 1  2  3  0  0
 1  2  3  4  0
 1  2  3  4  5
 1  2  3  4  5
 1  2  3  4  5
```

`LinearAlgebra.diagind` – Function.

```
diagind(M::AbstractMatrix, k::Integer = 0, indstyle::IndexStyle = IndexLinear())
diagind(M::AbstractMatrix, indstyle::IndexStyle = IndexLinear())
diagind(::IndexStyle, m::Integer, n::Integer, k::Integer = 0)
diagind(m::Integer, n::Integer, k::Integer = 0)
```

An `AbstractRange` giving the indices of the kth diagonal of a matrix, specified either by the matrix `M` itself or by its dimensions `m` and `n`. Optionally, an index style may be specified which determines the type of the range returned. If `indstyle isa IndexLinear` (default), this returns an

`AbstractRange{Integer}`. On the other hand, if `indstyle isa IndexCartesian`, this returns an `AbstractRange{CartesianIndex{2}}`.

If k is not provided, it is assumed to be `0` (corresponding to the main diagonal).

See also: `diag`, `diagm`, `Diagonal`.

Examples

The matrix itself may be passed to `diagind`:

```
julia> A = [1 2 3; 4 5 6; 7 8 9]
3×3 Matrix{Int64}:
 1  2  3
 4  5  6
 7  8  9

julia> diagind(A, -1)
2:4:6

julia> diagind(A, IndexCartesian())
StepRangeLen(CartesianIndex(1, 1), CartesianIndex(1, 1), 3)
```

Alternatively, dimensions `m` and `n` may be passed to get the diagonal of an `m×n` matrix:

```
julia> m, n = 5, 7
(5, 7)

julia> diagind(m, n, 2)
11:6:35

julia> diagind(IndexCartesian(), m, n)
StepRangeLen(CartesianIndex(1, 1), CartesianIndex(1, 1), 5)
```

Julia 1.11

Specifying an `IndexStyle` requires at least Julia 1.11.

`LinearAlgebra.diag` – Function.

```
diag(M, k::Integer=0)
```

The kth diagonal of a matrix, as a vector.

See also `diagm`, `diagind`, `Diagonal`, `isdiag`.

Examples

```
julia> A = [1 2 3; 4 5 6; 7 8 9]
3×3 Matrix{Int64}:
 1  2  3
 4  5  6
 7  8  9

julia> diag(A,1)
```

```
2-element Vector{Int64}:
 2
 6
```

`LinearAlgebra.diagm` – Function.

```
diagm(kv::Pair{<:Integer,<:AbstractVector}...)
diagm(m::Integer, n::Integer, kv::Pair{<:Integer,<:AbstractVector}...)
```

Construct a matrix from `Pairs` of diagonals and vectors. Vector `kv.second` will be placed on the `kv.first` diagonal. By default the matrix is square and its size is inferred from `kv`, but a non-square size `m×n` (padded with zeros as needed) can be specified by passing `m,n` as the first arguments. For repeated diagonal indices `kv.first` the values in the corresponding vectors `kv.second` will be added.

`diagm` constructs a full matrix; if you want storage-efficient versions with fast arithmetic, see `Diagonal`, `Bidiagonal` `Tridiagonal` and `SymTridiagonal`.

Examples

```
julia> diagm(1 => [1,2,3])
4×4 Matrix{Int64}:
 0  1  0  0
 0  0  2  0
 0  0  0  3
 0  0  0  0

julia> diagm(1 => [1,2,3], -1 => [4,5])
4×4 Matrix{Int64}:
 0  1  0  0
 4  0  2  0
 0  5  0  3
 0  0  0  0

julia> diagm(1 => [1,2,3], 1 => [1,2,3])
4×4 Matrix{Int64}:
 0  2  0  0
 0  0  4  0
 0  0  0  6
 0  0  0  0
```

```
diagm(v::AbstractVector)
diagm(m::Integer, n::Integer, v::AbstractVector)
```

Construct a matrix with elements of the vector as diagonal elements. By default, the matrix is square and its size is given by `length(v)`, but a non-square size `m×n` can be specified by passing `m,n` as the first arguments. The diagonal will be zero-padded if necessary.

Examples

```
julia> diagm([1,2,3])
3×3 Matrix{Int64}:
 1  0  0
 0  2  0
 0  0  3
```

```
julia> diagm(4, 5, [1,2,3])
4×5 Matrix{Int64}:
 1  0  0  0  0
 0  2  0  0  0
 0  0  3  0  0
 0  0  0  0  0
```

`LinearAlgebra.rank` – Function.

```
rank(::QRSparse{Tv,Ti}) -> Ti
```

Return the rank of the QR factorization

```
rank(S::SparseMatrixCSC{Tv,Ti}; [tol::Real]) -> Ti
```

Calculate rank of S by calculating its QR factorization. Values smaller than `tol` are considered as zero. See SPQR's manual.

```
rank(A::AbstractMatrix; atol::Real=0, rtol::Real=atol>0 ? 0 : n*ϵ)
rank(A::AbstractMatrix, rtol::Real)
```

Compute the numerical rank of a matrix by counting how many outputs of `svdvals(A)` are greater than `max(atol, rtol*σ₁)` where `σ₁` is `A`'s largest calculated singular value. `atol` and `rtol` are the absolute and relative tolerances, respectively. The default relative tolerance is `n*ϵ`, where `n` is the size of the smallest dimension of `A`, and `ϵ` is the `eps` of the element type of `A`.

Note

Numerical rank can be a sensitive and imprecise characterization of ill-conditioned matrices with singular values that are close to the threshold tolerance `max(atol, rtol*σ₁)`. In such cases, slight perturbations to the singular-value computation or to the matrix can change the result of rank by pushing one or more singular values across the threshold. These variations can even occur due to changes in floating-point errors between different Julia versions, architectures, compilers, or operating systems.

Julia 1.1

The `atol` and `rtol` keyword arguments requires at least Julia 1.1. In Julia 1.0 `rtol` is available as a positional argument, but this will be deprecated in Julia 2.0.

Examples

```
julia> rank(Matrix(I, 3, 3))
3

julia> rank(diagm(0 => [1, 0, 2]))
2

julia> rank(diagm(0 => [1, 0.001, 2]), rtol=0.1)
2
```

```
julia> rank(diagm(0 => [1, 0.001, 2]), rtol=0.00001)
3

julia> rank(diagm(0 => [1, 0.001, 2]), atol=1.5)
1
```

```
rank(S::SVD{<:Any, T}; atol::Real=0, rtol::Real=min(n,m)*ϵ) where {T}
```

Compute the numerical rank of the SVD object S by counting how many singular values are greater than `max(atol, rtol*σ₁)` where σ₁ is the largest calculated singular value. This is equivalent to the default `rank(::AbstractMatrix)` method except that it re-uses an existing SVD factorization. `atol` and `rtol` are the absolute and relative tolerances, respectively. The default relative tolerance is `n*ϵ`, where `n` is the size of the smallest dimension of `UΣV'` and `ϵ` is the `eps` of the element type of `S`.

Julia 1.12

The `rank(::SVD)` method requires at least Julia 1.12.

```
rank(A::QRPivoted{<:Any, T}; atol::Real=0, rtol::Real=min(n,m)*ϵ) where {T}
```

Compute the numerical rank of the QR factorization `A` by counting how many diagonal entries of `A.factors` are greater than `max(atol, rtol*Δ₁)` where Δ₁ is the largest calculated such entry. This is similar to the `rank(::AbstractMatrix)` method insofar as it counts the number of (numerically) nonzero coefficients from a matrix factorization, although the default method uses an SVD instead of a QR factorization. Like `rank(::SVD)`, this method also re-uses an existing matrix factorization.

Using a QR factorization to compute rank should typically produce the same result as using SVD, although it may be more prone to overestimating the rank in pathological cases where the matrix is ill-conditioned. It is also worth noting that it is generally faster to compute a QR factorization than it is to compute an SVD, so this method may be preferred when performance is a concern.

`atol` and `rtol` are the absolute and relative tolerances, respectively. The default relative tolerance is `n*ϵ`, where `n` is the size of the smallest dimension of `A` and `ϵ` is the `eps` of the element type of `A`.

Julia 1.12

The `rank(::QRPivoted)` method requires at least Julia 1.12.

`LinearAlgebra.norm` – Function.

```
norm(A, p::Real=2)
```

For any iterable container `A` (including arrays of any dimension) of numbers (or any element type for which `norm` is defined), compute the p-norm (defaulting to p=2) as if `A` were a vector of the corresponding length.

The p-norm is defined as

$$\|A\|_p = \left(\sum_{i=1}^{n} |a_i|^p \right)^{1/p}$$

with a_i the entries of A, $|a_i|$ the `norm` of a_i, and n the length of A. Since the p-norm is computed using the `norm`s of the entries of A, the p-norm of a vector of vectors is not compatible with the interpretation of it as a block vector in general if `p != 2`.

p can assume any numeric value (even though not all values produce a mathematically valid vector norm). In particular, `norm(A, Inf)` returns the largest value in `abs.(A)`, whereas `norm(A, -Inf)` returns the smallest. If A is a matrix and p=2, then this is equivalent to the Frobenius norm.

The second argument p is not necessarily a part of the interface for `norm`, i.e. a custom type may only implement `norm(A)` without second argument.

Use `opnorm` to compute the operator norm of a matrix.

Examples

```
julia> v = [3, -2, 6]
3-element Vector{Int64}:
  3
 -2
  6

julia> norm(v)
7.0

julia> norm(v, 1)
11.0

julia> norm(v, Inf)
6.0

julia> norm([1 2 3; 4 5 6; 7 8 9])
16.881943016134134

julia> norm([1 2 3 4 5 6 7 8 9])
16.881943016134134

julia> norm(1:9)
16.881943016134134

julia> norm(hcat(v,v), 1) == norm(vcat(v,v), 1) != norm([v,v], 1)
true

julia> norm(hcat(v,v), 2) == norm(vcat(v,v), 2) == norm([v,v], 2)
true

julia> norm(hcat(v,v), Inf) == norm(vcat(v,v), Inf) != norm([v,v], Inf)
true
```

```
norm(x::Number, p::Real=2)
```

For numbers, return $\left(|x|^p\right)^{1/p}$.

Examples

```
julia> norm(2, 1)
```

```
2.0

julia> norm(-2, 1)
2.0

julia> norm(2, 2)
2.0

julia> norm(-2, 2)
2.0

julia> norm(2, Inf)
2.0

julia> norm(-2, Inf)
2.0
```

`LinearAlgebra.opnorm` – Function.

```
opnorm(A::AbstractMatrix, p::Real=2)
```

Compute the operator norm (or matrix norm) induced by the vector p-norm, where valid values of p are 1, 2, or `Inf`. (Note that for sparse matrices, p=2 is currently not implemented.) Use `norm` to compute the Frobenius norm.

When p=1, the operator norm is the maximum absolute column sum of A:

$$\|A\|_1 = \max_{1jn} \sum_{i=1}^{m} |a_{ij}|$$

with a_{ij} the entries of A, and m and n its dimensions.

When p=2, the operator norm is the spectral norm, equal to the largest singular value of A.

When p=`Inf`, the operator norm is the maximum absolute row sum of A:

$$\|A\|_\infty = \max_{1im} \sum_{j=1}^{n} |a_{ij}|$$

Examples

```
julia> A = [1 -2 -3; 2 3 -1]
2×3 Matrix{Int64}:
 1  -2  -3
 2   3  -1

julia> opnorm(A, Inf)
6.0

julia> opnorm(A, 1)
5.0
```

```
opnorm(x::Number, p::Real=2)
```

For numbers, return $\left(|x|^p\right)^{1/p}$. This is equivalent to `norm`.

```
opnorm(A::Adjoint{<:Any,<:AbstractVector}, q::Real=2)
opnorm(A::Transpose{<:Any,<:AbstractVector}, q::Real=2)
```

For Adjoint/Transpose-wrapped vectors, return the operator q-norm of `A`, which is equivalent to the p-norm with value `p = q/(q-1)`. They coincide at `p = q = 2`. Use `norm` to compute the p norm of `A` as a vector.

The difference in norm between a vector space and its dual arises to preserve the relationship between duality and the dot product, and the result is consistent with the operator p-norm of a `1 × n` matrix.

Examples

```
julia> v = [1; im];

julia> vc = v';

julia> opnorm(vc, 1)
1.0

julia> norm(vc, 1)
2.0

julia> norm(v, 1)
2.0

julia> opnorm(vc, 2)
1.4142135623730951

julia> norm(vc, 2)
1.4142135623730951

julia> norm(v, 2)
1.4142135623730951

julia> opnorm(vc, Inf)
2.0

julia> norm(vc, Inf)
1.0

julia> norm(v, Inf)
1.0
```

`LinearAlgebra.normalize!` – Function.

```
normalize!(a::AbstractArray, p::Real=2)
```

Normalize the array `a` in-place so that its p-norm equals unity, i.e. `norm(a, p) == 1`. See also `normalize` and `norm`.

`LinearAlgebra.normalize` – Function.

```
normalize(a, p::Real=2)
```

Normalize a so that its p-norm equals unity, i.e. `norm(a, p) == 1`. For scalars, this is similar to sign(a), except normalize(0) = NaN. See also `normalize!`, `norm`, and `sign`.

Examples

```
julia> a = [1,2,4];

julia> b = normalize(a)
3-element Vector{Float64}:
 0.2182178902359924
 0.4364357804719848
 0.8728715609439696

julia> norm(b)
1.0

julia> c = normalize(a, 1)
3-element Vector{Float64}:
 0.14285714285714285
 0.2857142857142857
 0.5714285714285714

julia> norm(c, 1)
1.0

julia> a = [1 2 4 ; 1 2 4]
2×3 Matrix{Int64}:
 1  2  4
 1  2  4

julia> norm(a)
6.48074069840786

julia> normalize(a)
2×3 Matrix{Float64}:
 0.154303  0.308607  0.617213
 0.154303  0.308607  0.617213

julia> normalize(3, 1)
1.0

julia> normalize(-8, 1)
-1.0

julia> normalize(0, 1)
NaN
```

`LinearAlgebra.cond` – Function.

```
cond(M, p::Real=2)
```

Condition number of the matrix `M`, computed using the operator p-norm. Valid values for `p` are `1`, `2` (default), or `Inf`.

`LinearAlgebra.condskeel` – Function.

```
condskeel(M, [x, p::Real=Inf])
```

$$\kappa_S(M, p) = \left\| |M| \left| M^{-1} \right| \right\|_p$$
$$\kappa_S(M, x, p) = \frac{\left\| |M| \left| M^{-1} \right| |x| \right\|_p}{\|x\|_p}$$

Skeel condition number κ_S of the matrix `M`, optionally with respect to the vector `x`, as computed using the operator p-norm. $|M|$ denotes the matrix of (entry wise) absolute values of M; $|M|_{ij} = |M_{ij}|$. Valid values for `p` are `1`, `2` and `Inf` (default).

This quantity is also known in the literature as the Bauer condition number, relative condition number, or componentwise relative condition number.

`LinearAlgebra.tr` – Function.

```
tr(M)
```

Matrix trace. Sums the diagonal elements of `M`.

Examples

```
julia> A = [1 2; 3 4]
2×2 Matrix{Int64}:
 1  2
 3  4

julia> tr(A)
5
```

`LinearAlgebra.det` – Function.

```
det(M)
```

Matrix determinant.

See also: `logdet` and `logabsdet`.

Examples

```
julia> M = [1 0; 2 2]
2×2 Matrix{Int64}:
 1  0
 2  2

julia> det(M)
2.0
```

Note that, in general, `det` computes a floating-point approximation of the determinant, even for integer matrices, typically via Gaussian elimination. Julia includes an exact algorithm for integer determinants (the Bareiss algorithm), but only uses it by default for `BigInt` matrices (since determinants quickly overflow any fixed integer precision):

```
julia> det(BigInt[1 0; 2 2]) # exact integer determinant
2
```

`LinearAlgebra.logdet` – Function.

```
logdet(M)
```

Logarithm of matrix determinant. Equivalent to `log(det(M))`, but may provide increased accuracy and avoids overflow/underflow.

Examples

```
julia> M = [1 0; 2 2]
2×2 Matrix{Int64}:
 1  0
 2  2

julia> logdet(M)
0.6931471805599453

julia> logdet(Matrix(I, 3, 3))
0.0
```

`LinearAlgebra.logabsdet` – Function.

```
logabsdet(M)
```

Log of absolute value of matrix determinant. Equivalent to `(log(abs(det(M))), sign(det(M)))`, but may provide increased accuracy and/or speed.

Examples

```
julia> A = [-1. 0.; 0. 1.]
2×2 Matrix{Float64}:
 -1.0  0.0
  0.0  1.0

julia> det(A)
-1.0

julia> logabsdet(A)
(0.0, -1.0)

julia> B = [2. 0.; 0. 1.]
2×2 Matrix{Float64}:
 2.0  0.0
 0.0  1.0
```

```
julia> det(B)
2.0

julia> logabsdet(B)
(0.6931471805599453, 1.0)
```

Base.inv – Method.

```
inv(M)
```

Matrix inverse. Computes matrix N such that `M * N = I`, where I is the identity matrix. Computed by solving the left-division `N = M \ I`.

A SingularException is thrown if M fails numerical inversion.

Examples

```
julia> M = [2 5; 1 3]
2×2 Matrix{Int64}:
 2  5
 1  3

julia> N = inv(M)
2×2 Matrix{Float64}:
  3.0  -5.0
 -1.0   2.0

julia> M*N == N*M == Matrix(I, 2, 2)
true
```

LinearAlgebra.pinv – Function.

```
pinv(M; atol::Real=0, rtol::Real=atol>0 ? 0 : n*ϵ)
pinv(M, rtol::Real) = pinv(M; rtol=rtol) # to be deprecated in Julia 2.0
```

Computes the Moore-Penrose pseudoinverse.

For matrices M with floating point elements, it is convenient to compute the pseudoinverse by inverting only singular values greater than `max(atol, rtol*σ₁)` where σ_1 is the largest singular value of M.

The optimal choice of absolute (`atol`) and relative tolerance (`rtol`) varies both with the value of M and the intended application of the pseudoinverse. The default relative tolerance is `n*ϵ`, where n is the size of the smallest dimension of M, and ϵ is the eps of the element type of M.

For solving dense, ill-conditioned equations in a least-square sense, it is better to *not* explicitly form the pseudoinverse matrix, since this can lead to numerical instability at low tolerances. The default `M \ b` algorithm instead uses pivoted QR factorization (qr). To use an SVD-based algorithm, it is better to employ the SVD directly via `svd(M; rtol, atol) \ b` or `ldiv!(svd(M), b; rtol, atol)`.

One can also pass `M = svd(A)` as the argument to pinv in order to re-use an existing SVD factorization. In this case, pinv will return the SVD of the pseudo-inverse, which can be applied accurately, instead of an explicit matrix.

Julia 1.13

Passing an SVD object to pinv requires Julia 1.13 or later.

For more information, see [8], [9], [10], [11].

Examples

```
julia> M = [1.5 1.3; 1.2 1.9]
2×2 Matrix{Float64}:
 1.5  1.3
 1.2  1.9

julia> N = pinv(M)
2×2 Matrix{Float64}:
  1.47287   -1.00775
 -0.930233   1.16279

julia> M * N
2×2 Matrix{Float64}:
 1.0          -2.22045e-16
 4.44089e-16   1.0
```

`LinearAlgebra.nullspace` – Function.

```
nullspace(M; atol::Real=0, rtol::Real=atol>0 ? 0 : n*ϵ)
nullspace(M, rtol::Real) = nullspace(M; rtol=rtol) # to be deprecated in Julia 2.0
```

Computes a basis for the nullspace of `M` by including the singular vectors of `M` whose singular values have magnitudes smaller than `max(atol, rtol*σ₁)`, where `σ₁` is `M`'s largest singular value.

By default, the relative tolerance `rtol` is `n*ϵ`, where `n` is the size of the smallest dimension of `M`, and `ϵ` is the `eps` of the element type of `M`.

Examples

```
julia> M = [1 0 0; 0 1 0; 0 0 0]
3×3 Matrix{Int64}:
 1  0  0
 0  1  0
 0  0  0

julia> nullspace(M)
3×1 Matrix{Float64}:
 0.0
 0.0
 1.0
```

[8] PR 1387, "stable pinv least-squares", LinearAlgebra.jl#1387

[9] Åke Björck, "Numerical Methods for Least Squares Problems", SIAM Press, Philadelphia, 1996, "Other Titles in Applied Mathematics", Vol. 51. doi:10.1137/1.9781611971484

[10] G. W. Stewart, "Rank Degeneracy", SIAM Journal on Scientific and Statistical Computing, 5(2), 1984, 403-413. doi:10.1137/0905030

[11] Konstantinos Konstantinides and Kung Yao, "Statistical analysis of effective singular values in matrix rank determination", IEEE Transactions on Acoustics, Speech and Signal Processing, 36(5), 1988, 757-763. doi:10.1109/29.1585

```
julia> nullspace(M, rtol=3)
3×3 Matrix{Float64}:
 0.0  1.0  0.0
 1.0  0.0  0.0
 0.0  0.0  1.0

julia> nullspace(M, atol=0.95)
3×1 Matrix{Float64}:
 0.0
 0.0
 1.0
```

Base.kron – Function.

```
kron(A, B)
```

Computes the Kronecker product of two vectors, matrices or numbers.

For real vectors v and w, the Kronecker product is related to the outer product by `kron(v,w) == vec(w * transpose(v))` or `w * transpose(v) == reshape(kron(v,w), (length(w), length(v)))`. Note how the ordering of v and w differs on the left and right of these expressions (due to column-major storage). For complex vectors, the outer product `w * v'` also differs by conjugation of v.

Examples

```
julia> A = [1 2; 3 4]
2×2 Matrix{Int64}:
 1  2
 3  4

julia> B = [im 1; 1 -im]
2×2 Matrix{Complex{Int64}}:
 0+1im  1+0im
 1+0im  0-1im

julia> kron(A, B)
4×4 Matrix{Complex{Int64}}:
 0+1im  1+0im  0+2im  2+0im
 1+0im  0-1im  2+0im  0-2im
 0+3im  3+0im  0+4im  4+0im
 3+0im  0-3im  4+0im  0-4im

julia> v = [1, 2]; w = [3, 4, 5];

julia> w*transpose(v)
3×2 Matrix{Int64}:
 3   6
 4   8
 5  10

julia> reshape(kron(v,w), (length(w), length(v)))
3×2 Matrix{Int64}:
 3   6
```

```
4   8
5  10
```

`Base.kron!` – Function.

```
kron!(C, A, B)
```

Computes the Kronecker product of `A` and `B` and stores the result in `C`, overwriting the existing content of `C`. This is the in-place version of `kron`.

Julia 1.6

This function requires Julia 1.6 or later.

`Base.exp` – Method.

```
exp(A::AbstractMatrix)
```

Compute the matrix exponential of `A`, defined by

$$e^A = \sum_{n=0}^{\infty} \frac{A^n}{n!}.$$

For symmetric or Hermitian `A`, an eigendecomposition (`eigen`) is used, otherwise the scaling and squaring algorithm (see [12]) is chosen.

Examples

```
julia> A = Matrix(1.0I, 2, 2)
2×2 Matrix{Float64}:
 1.0  0.0
 0.0  1.0

julia> exp(A)
2×2 Matrix{Float64}:
 2.71828  0.0
 0.0      2.71828
```

`Base.cis` – Method.

```
cis(A::AbstractMatrix)
```

More efficient method for `exp(im*A)` of square matrix `A` (especially if `A` is `Hermitian` or real-`Symmetric`).

See also `cispi`, `sincos`, `exp`.

Julia 1.7

Support for using `cis` with matrices was added in Julia 1.7.

Examples

[12] Nicholas J. Higham, "The squaring and scaling method for the matrix exponential revisited", SIAM Journal on Matrix Analysis and Applications, 26(4), 2005, 1179-1193. doi:10.1137/090768539

```
julia> cis([π 0; 0 π]) ≈ -I
true
```

Base.:^ – Method.

```
^(A::AbstractMatrix, p::Number)
```

Matrix power, equivalent to $\exp(p \log(A))$

Examples

```
julia> [1 2; 0 3]^3
2×2 Matrix{Int64}:
 1  26
 0  27
```

Base.:^ – Method.

```
^(b::Number, A::AbstractMatrix)
```

Matrix exponential, equivalent to $\exp(\log(b)A)$.

Julia 1.1

Support for raising `Irrational` numbers (like □) to a matrix was added in Julia 1.1.

Examples

```
julia> 2^[1 2; 0 3]
2×2 Matrix{Float64}:
 2.0  6.0
 0.0  8.0

julia> □^[1 2; 0 3]
2×2 Matrix{Float64}:
 2.71828  17.3673
 0.0      20.0855
```

Base.log – Method.

```
log(A::AbstractMatrix)
```

If A has no negative real eigenvalue, compute the principal matrix logarithm of A, i.e. the unique matrix X such that $e^X = A$ and $-\pi < Im(\lambda) < \pi$ for all the eigenvalues λ of X. If A has nonpositive eigenvalues, a nonprincipal matrix function is returned whenever possible.

If A is symmetric or Hermitian, its eigendecomposition (eigen) is used, if A is triangular an improved version of the inverse scaling and squaring method is employed (see [13] and [14]). If A is real with no negative eigenvalues, then the real Schur form is computed. Otherwise, the complex Schur form is

computed. Then the upper (quasi-)triangular algorithm in [14] is used on the upper (quasi-)triangular factor.

Examples

```
julia> A = Matrix(2.7182818*I, 2, 2)
2×2 Matrix{Float64}:
 2.71828  0.0
 0.0      2.71828

julia> log(A)
2×2 Matrix{Float64}:
 1.0  0.0
 0.0  1.0
```

`Base.sqrt` – Method.

```
sqrt(A::AbstractMatrix)
```

If A has no negative real eigenvalues, compute the principal matrix square root of A, that is the unique matrix X with eigenvalues having positive real part such that $X^2 = A$. Otherwise, a nonprincipal square root is returned.

If A is real-symmetric or Hermitian, its eigendecomposition (`eigen`) is used to compute the square root. For such matrices, eigenvalues λ that appear to be slightly negative due to roundoff errors are treated as if they were zero. More precisely, matrices with all eigenvalues ≥ `-rtol*(max |λ|)` are treated as semidefinite (yielding a Hermitian square root), with negative eigenvalues taken to be zero. `rtol` is a keyword argument to `sqrt` (in the Hermitian/real-symmetric case only) that defaults to machine precision scaled by `size(A,1)`.

Otherwise, the square root is determined by means of the Björck-Hammarling method [15], which computes the complex Schur form (`schur`) and then the complex square root of the triangular factor. If a real square root exists, then an extension of this method [16] that computes the real Schur form and then the real square root of the quasi-triangular factor is instead used.

Examples

```
julia> A = [4 0; 0 4]
2×2 Matrix{Int64}:
 4  0
 0  4
```

[13] Awad H. Al-Mohy and Nicholas J. Higham, "Improved inverse scaling and squaring algorithms for the matrix logarithm", SIAM Journal on Scientific Computing, 34(4), 2012, C153-C169. doi:10.1137/110852553

[14] Awad H. Al-Mohy, Nicholas J. Higham and Samuel D. Relton, "Computing the Fréchet derivative of the matrix logarithm and estimating the condition number", SIAM Journal on Scientific Computing, 35(4), 2013, C394-C410. doi:10.1137/120885991

[15] Åke Björck and Sven Hammarling, "A Schur method for the square root of a matrix", Linear Algebra and its Applications, 52-53, 1983, 127-140. doi:10.1016/0024-3795(83)80010-X

[16] Nicholas J. Higham, "Computing real square roots of a real matrix", Linear Algebra and its Applications, 88-89, 1987, 405-430. doi:10.1016/0024-3795(87)90118-2

```
julia> sqrt(A)
2×2 Matrix{Float64}:
 2.0  0.0
 0.0  2.0
```

Base.Math.cbrt – Method.

```
cbrt(A::AbstractMatrix{<:Real})
```

Computes the real-valued cube root of a real-valued matrix A. If `T = cbrt(A)`, then we have `T*T*T ≈ A`, see example given below.

If A is real-symmetric or Hermitian, its eigendecomposition (eigen) is used to find the cube root. Otherwise, a specialized version of the p-th root algorithm [17] is utilized, which exploits the real-valued Schur decomposition (schur) to compute the cube root.

Examples

```
julia> A = [0.927524 -0.15857; -1.3677 -1.01172]
2×2 Matrix{Float64}:
  0.927524  -0.15857
 -1.3677    -1.01172

julia> T = cbrt(A)
2×2 Matrix{Float64}:
  0.910077  -0.151019
 -1.30257   -0.936818

julia> T*T*T ≈ A
true
```

Base.cos – Method.

```
cos(A::AbstractMatrix)
```

Compute the matrix cosine of a square matrix A.

If A is real-symmetric or Hermitian, its eigendecomposition (eigen) is used to compute the cosine. Otherwise, the cosine is determined by calling exp.

Examples

```
julia> cos(fill(1.0, (2,2)))
2×2 Matrix{Float64}:
  0.291927  -0.708073
 -0.708073   0.291927
```

Base.sin – Method.

```
sin(A::AbstractMatrix)
```

[17] Matthew I. Smith, "A Schur Algorithm for Computing Matrix pth Roots", SIAM Journal on Matrix Analysis and Applications, vol. 24, 2003, pp. 971–989. doi:10.1137/S0895479801392697

Compute the matrix sine of a square matrix A.

If A is real-symmetric or Hermitian, its eigendecomposition (eigen) is used to compute the sine. Otherwise, the sine is determined by calling exp.

Examples

```
julia> sin(fill(1.0, (2,2)))
2×2 Matrix{Float64}:
 0.454649  0.454649
 0.454649  0.454649
```

Base.Math.sincos – Method.

```
sincos(A::AbstractMatrix)
```

Compute the matrix sine and cosine of a square matrix A.

Examples

```
julia> S, C = sincos(fill(1.0, (2,2)));

julia> S
2×2 Matrix{Float64}:
 0.454649  0.454649
 0.454649  0.454649

julia> C
2×2 Matrix{Float64}:
  0.291927  -0.708073
 -0.708073   0.291927
```

Base.tan – Method.

```
tan(A::AbstractMatrix)
```

Compute the matrix tangent of a square matrix A.

If A is real-symmetric or Hermitian, its eigendecomposition (eigen) is used to compute the tangent. Otherwise, the tangent is determined by calling exp.

Examples

```
julia> tan(fill(1.0, (2,2)))
2×2 Matrix{Float64}:
 -1.09252  -1.09252
 -1.09252  -1.09252
```

Base.Math.sec – Method.

```
sec(A::AbstractMatrix)
```

Compute the matrix secant of a square matrix A.

Base.Math.csc – Method.

```
csc(A::AbstractMatrix)
```

Compute the matrix cosecant of a square matrix A.

Base.Math.cot – Method.

```
cot(A::AbstractMatrix)
```

Compute the matrix cotangent of a square matrix A.

Base.cosh – Method.

```
cosh(A::AbstractMatrix)
```

Compute the matrix hyperbolic cosine of a square matrix A.

Base.sinh – Method.

```
sinh(A::AbstractMatrix)
```

Compute the matrix hyperbolic sine of a square matrix A.

Base.tanh – Method.

```
tanh(A::AbstractMatrix)
```

Compute the matrix hyperbolic tangent of a square matrix A.

Base.Math.sech – Method.

```
sech(A::AbstractMatrix)
```

Compute the matrix hyperbolic secant of square matrix A.

Base.Math.csch – Method.

```
csch(A::AbstractMatrix)
```

Compute the matrix hyperbolic cosecant of square matrix A.

Base.Math.coth – Method.

```
coth(A::AbstractMatrix)
```

Compute the matrix hyperbolic cotangent of square matrix A.

Base.acos – Method.

```
acos(A::AbstractMatrix)
```

Compute the inverse matrix cosine of a square matrix A.

If A is real-symmetric or Hermitian, its eigendecomposition (eigen) is used to compute the inverse cosine. Otherwise, the inverse cosine is determined by using log and sqrt. For the theory and logarithmic formulas used to compute this function, see [18].

Examples

[18] Mary Aprahamian and Nicholas J. Higham, "Matrix Inverse Trigonometric and Inverse Hyperbolic Functions: Theory and Algorithms", MIMS EPrint: 2016.4. https://doi.org/10.1137/16M1057577

```
julia> acos(cos([0.5 0.1; -0.2 0.3]))
2×2 Matrix{ComplexF64}:
  0.5-8.32667e-17im  0.1+0.0im
 -0.2+2.63678e-16im  0.3-3.46945e-16im
```

Base.asin – Method.

```
asin(A::AbstractMatrix)
```

Compute the inverse matrix sine of a square matrix A.

If A is real-symmetric or Hermitian, its eigendecomposition (eigen) is used to compute the inverse sine. Otherwise, the inverse sine is determined by using log and sqrt. For the theory and logarithmic formulas used to compute this function, see [19].

Examples

```
julia> asin(sin([0.5 0.1; -0.2 0.3]))
2×2 Matrix{ComplexF64}:
  0.5-4.16334e-17im  0.1-5.55112e-17im
 -0.2+9.71445e-17im  0.3-1.249e-16im
```

Base.atan – Method.

```
atan(A::AbstractMatrix)
```

Compute the inverse matrix tangent of a square matrix A.

If A is real-symmetric or Hermitian, its eigendecomposition (eigen) is used to compute the inverse tangent. Otherwise, the inverse tangent is determined by using log. For the theory and logarithmic formulas used to compute this function, see [20].

Examples

```
julia> atan(tan([0.5 0.1; -0.2 0.3]))
2×2 Matrix{ComplexF64}:
  0.5  0.1
 -0.2  0.3
```

Base.Math.asec – Method.

```
asec(A::AbstractMatrix)
```

Compute the inverse matrix secant of A.

Base.Math.acsc – Method.

```
acsc(A::AbstractMatrix)
```

[19] Mary Aprahamian and Nicholas J. Higham, "Matrix Inverse Trigonometric and Inverse Hyperbolic Functions: Theory and Algorithms", MIMS EPrint: 2016.4. https://doi.org/10.1137/16M1057577

[20] Mary Aprahamian and Nicholas J. Higham, "Matrix Inverse Trigonometric and Inverse Hyperbolic Functions: Theory and Algorithms", MIMS EPrint: 2016.4. https://doi.org/10.1137/16M1057577

Compute the inverse matrix cosecant of A.

Base.Math.acot – Method.

```
acot(A::AbstractMatrix)
```

Compute the inverse matrix cotangent of A.

Base.acosh – Method.

```
acosh(A::AbstractMatrix)
```

Compute the inverse hyperbolic matrix cosine of a square matrix A. For the theory and logarithmic formulas used to compute this function, see [21].

Base.asinh – Method.

```
asinh(A::AbstractMatrix)
```

Compute the inverse hyperbolic matrix sine of a square matrix A. For the theory and logarithmic formulas used to compute this function, see [22].

Base.atanh – Method.

```
atanh(A::AbstractMatrix)
```

Compute the inverse hyperbolic matrix tangent of a square matrix A. For the theory and logarithmic formulas used to compute this function, see [23].

Base.Math.asech – Method.

```
asech(A::AbstractMatrix)
```

Compute the inverse matrix hyperbolic secant of A.

Base.Math.acsch – Method.

```
acsch(A::AbstractMatrix)
```

Compute the inverse matrix hyperbolic cosecant of A.

Base.Math.acoth – Method.

```
acoth(A::AbstractMatrix)
```

Compute the inverse matrix hyperbolic cotangent of A.

[21] Mary Aprahamian and Nicholas J. Higham, "Matrix Inverse Trigonometric and Inverse Hyperbolic Functions: Theory and Algorithms", MIMS EPrint: 2016.4. https://doi.org/10.1137/16M1057577

[22] Mary Aprahamian and Nicholas J. Higham, "Matrix Inverse Trigonometric and Inverse Hyperbolic Functions: Theory and Algorithms", MIMS EPrint: 2016.4. https://doi.org/10.1137/16M1057577

[23] Mary Aprahamian and Nicholas J. Higham, "Matrix Inverse Trigonometric and Inverse Hyperbolic Functions: Theory and Algorithms", MIMS EPrint: 2016.4. https://doi.org/10.1137/16M1057577

`LinearAlgebra.lyap` – Function.

```
lyap(A, C)
```

Computes the solution `X` to the continuous Lyapunov equation `AX + XA' + C = 0`, where no eigenvalue of `A` has a zero real part and no two eigenvalues are negative complex conjugates of each other.

Examples

```
julia> A = [3. 4.; 5. 6]
2×2 Matrix{Float64}:
 3.0  4.0
 5.0  6.0

julia> B = [1. 1.; 1. 2.]
2×2 Matrix{Float64}:
 1.0  1.0
 1.0  2.0

julia> X = lyap(A, B)
2×2 Matrix{Float64}:
  0.5  -0.5
 -0.5   0.25

julia> A*X + X*A' ≈ -B
true
```

`LinearAlgebra.sylvester` – Function.

```
sylvester(A, B, C)
```

Computes the solution `X` to the Sylvester equation `AX + XB + C = 0`, where `A`, `B` and `C` have compatible dimensions and `A` and `-B` have no eigenvalues with equal real part.

Examples

```
julia> A = [3. 4.; 5. 6]
2×2 Matrix{Float64}:
 3.0  4.0
 5.0  6.0

julia> B = [1. 1.; 1. 2.]
2×2 Matrix{Float64}:
 1.0  1.0
 1.0  2.0

julia> C = [1. 2.; -2. 1]
2×2 Matrix{Float64}:
  1.0  2.0
 -2.0  1.0

julia> X = sylvester(A, B, C)
2×2 Matrix{Float64}:
 -4.46667   1.93333
```

```
 3.73333  -1.8

julia> A*X + X*B ≈ -C
true
```

`LinearAlgebra.issuccess` – Function.

```
issuccess(F::Factorization)
```

Test that a factorization of a matrix succeeded.

Julia 1.6

`issuccess(::CholeskyPivoted)` requires Julia 1.6 or later.

Examples

```
julia> F = cholesky([1 0; 0 1]);

julia> issuccess(F)
true
```

```
issuccess(F::LU; allowsingular = false)
```

Test that the LU factorization of a matrix succeeded. By default a factorization that produces a valid but rank-deficient U factor is considered a failure. This can be changed by passing `allowsingular = true`.

Julia 1.11

The `allowsingular` keyword argument was added in Julia 1.11.

Examples

```
julia> F = lu([1 2; 1 2], check = false);

julia> issuccess(F)
false

julia> issuccess(F, allowsingular = true)
true
```

`LinearAlgebra.issymmetric` – Function.

```
issymmetric(A) -> Bool
```

Test whether a matrix or number is symmetric.

Examples

```
julia> a = [1 2; 2 -1]
2×2 Matrix{Int64}:
 1   2
```

```
 2  -1

julia> issymmetric(a)
true

julia> b = [1 im; -im 1]
2×2 Matrix{Complex{Int64}}:
 1+0im  0+1im
 0-1im  1+0im

julia> issymmetric(b)
false
```

`LinearAlgebra.isposdef` – Function.

```
isposdef(A) -> Bool
```

Test whether a matrix is positive definite (and Hermitian) by trying to perform a Cholesky factorization of A.

See also `isposdef!`, `cholesky`.

Examples

```
julia> A = [1 2; 2 50]
2×2 Matrix{Int64}:
 1   2
 2  50

julia> isposdef(A)
true
```

`LinearAlgebra.isposdef!` – Function.

```
isposdef!(A) -> Bool
```

Test whether a matrix is positive definite (and Hermitian) by trying to perform a Cholesky factorization of A, overwriting A in the process. See also `isposdef`.

Examples

```
julia> A = [1. 2.; 2. 50.];

julia> isposdef!(A)
true

julia> A
2×2 Matrix{Float64}:
 1.0  2.0
 2.0  6.78233
```

`LinearAlgebra.istril` – Function.

```
istril(A::AbstractMatrix, k::Integer = 0) -> Bool
```

Test whether A is lower triangular starting from the kth superdiagonal.

Examples

```
julia> a = [1 2; 2 -1]
2×2 Matrix{Int64}:
 1   2
 2  -1

julia> istril(a)
false

julia> istril(a, 1)
true

julia> c = [1 1 0; 1 1 1; 1 1 1]
3×3 Matrix{Int64}:
 1  1  0
 1  1  1
 1  1  1

julia> istril(c)
false

julia> istril(c, 1)
true
```

`LinearAlgebra.istriu` – Function.

```
istriu(A::AbstractMatrix, k::Integer = 0) -> Bool
```

Test whether A is upper triangular starting from the kth superdiagonal.

Examples

```
julia> a = [1 2; 2 -1]
2×2 Matrix{Int64}:
 1   2
 2  -1

julia> istriu(a)
false

julia> istriu(a, -1)
true

julia> c = [1 1 1; 1 1 1; 0 1 1]
3×3 Matrix{Int64}:
 1  1  1
 1  1  1
 0  1  1

julia> istriu(c)
false
```

```
julia> istriu(c, -1)
true
```

`LinearAlgebra.isdiag` – Function.

```
isdiag(A) -> Bool
```

Test whether a matrix is diagonal in the sense that `iszero(A[i,j])` is true unless `i == j`. Note that it is not necessary for A to be square; if you would also like to check that, you need to check that `size(A, 1) == size(A, 2)`.

Examples

```
julia> a = [1 2; 2 -1]
2×2 Matrix{Int64}:
 1   2
 2  -1

julia> isdiag(a)
false

julia> b = [im 0; 0 -im]
2×2 Matrix{Complex{Int64}}:
 0+1im  0+0im
 0+0im  0-1im

julia> isdiag(b)
true

julia> c = [1 0 0; 0 2 0]
2×3 Matrix{Int64}:
 1  0  0
 0  2  0

julia> isdiag(c)
true

julia> d = [1 0 0; 0 2 3]
2×3 Matrix{Int64}:
 1  0  0
 0  2  3

julia> isdiag(d)
false
```

`LinearAlgebra.ishermitian` – Function.

```
ishermitian(A) -> Bool
```

Test whether a matrix is Hermitian.

Examples

```
julia> a = [1 2; 2 -1]
2×2 Matrix{Int64}:
 1   2
 2  -1

julia> ishermitian(a)
true

julia> b = [1 im; -im 1]
2×2 Matrix{Complex{Int64}}:
 1+0im  0+1im
 0-1im  1+0im

julia> ishermitian(b)
true
```

Base.transpose – Function.

```
transpose(A)
```

Lazy transpose. Mutating the returned object should appropriately mutate A. Often, but not always, yields Transpose(A), where Transpose is a lazy transpose wrapper. Note that this operation is recursive.

This operation is intended for linear algebra usage - for general data manipulation see permutedims, which is non-recursive.

Examples

```
julia> A = [3 2; 0 0]
2×2 Matrix{Int64}:
 3  2
 0  0

julia> B = transpose(A)
2×2 transpose(::Matrix{Int64}) with eltype Int64:
 3  0
 2  0

julia> B isa Transpose
true

julia> transpose(B) === A # the transpose of a transpose unwraps the parent
true

julia> Transpose(B) # however, the constructor always wraps its argument
2×2 transpose(transpose(::Matrix{Int64})) with eltype Int64:
 3  2
 0  0

julia> B[1,2] = 4; # modifying B will modify A automatically

julia> A
2×2 Matrix{Int64}:
```

```
3  2
4  0
```

For complex matrices, the `adjoint` operation is equivalent to a conjugate-transpose.

```
julia> A = reshape([Complex(x, x) for x in 1:4], 2, 2)
2×2 Matrix{Complex{Int64}}:
 1+1im  3+3im
 2+2im  4+4im

julia> adjoint(A) == conj(transpose(A))
true
```

The `transpose` of an `AbstractVector` is a row-vector:

```
julia> v = [1,2,3]
3-element Vector{Int64}:
 1
 2
 3

julia> transpose(v) # returns a row-vector
1×3 transpose(::Vector{Int64}) with eltype Int64:
 1  2  3

julia> transpose(v) * v # compute the dot product
14
```

For a matrix of matrices, the individual blocks are recursively operated on:

```
julia> C = [1 3; 2 4]
2×2 Matrix{Int64}:
 1  3
 2  4

julia> D = reshape([C, 2C, 3C, 4C], 2, 2) # construct a block matrix
2×2 Matrix{Matrix{Int64}}:
 [1 3; 2 4]  [3 9; 6 12]
 [2 6; 4 8]  [4 12; 8 16]

julia> transpose(D) # blocks are recursively transposed
2×2 transpose(::Matrix{Matrix{Int64}}) with eltype Transpose{Int64, Matrix{Int64}}:
 [1 2; 3 4]   [2 4; 6 8]
 [3 6; 9 12]  [4 8; 12 16]
```

```
transpose(F::Factorization)
```

Lazy transpose of the factorization F. By default, returns a `TransposeFactorization`, except for `Factorizations` with real `eltype`, in which case returns an `AdjointFactorization`.

`LinearAlgebra.transpose!` – Function.

```
transpose!(dest,src)
```

Transpose array `src` and store the result in the preallocated array `dest`, which should have a size corresponding to `(size(src,2),size(src,1))`. No in-place transposition is supported and unexpected results will happen if `src` and `dest` have overlapping memory regions.

Examples

```
julia> A = [3+2im 9+2im; 8+7im  4+6im]
2×2 Matrix{Complex{Int64}}:
 3+2im  9+2im
 8+7im  4+6im

julia> B = zeros(Complex{Int64}, 2, 2)
2×2 Matrix{Complex{Int64}}:
 0+0im  0+0im
 0+0im  0+0im

julia> transpose!(B, A);

julia> B
2×2 Matrix{Complex{Int64}}:
 3+2im  8+7im
 9+2im  4+6im

julia> A
2×2 Matrix{Complex{Int64}}:
 3+2im  9+2im
 8+7im  4+6im
```

```
transpose!(X::AbstractSparseMatrixCSC{Tv,Ti}, A::AbstractSparseMatrixCSC{Tv,Ti}) where {Tv,Ti}
```

Transpose the matrix A and stores it in the matrix X. `size(X)` must be equal to `size(transpose(A))`. No additional memory is allocated other than resizing the rowval and nzval of X, if needed.

See `halfperm!`

`LinearAlgebra.Transpose` – Type.

```
Transpose
```

Lazy wrapper type for a transpose view of the underlying linear algebra object, usually an `AbstractVector`/`AbstractMatrix`. Usually, the `Transpose` constructor should not be called directly, use `transpose` instead. To materialize the view use `copy`.

This type is intended for linear algebra usage - for general data manipulation see `permutedims`.

Examples

```
julia> A = [2 3; 0 0]
2×2 Matrix{Int64}:
 2  3
 0  0

julia> Transpose(A)
2×2 transpose(::Matrix{Int64}) with eltype Int64:
 2  0
 3  0
```

`LinearAlgebra.TransposeFactorization` – Type.

```
TransposeFactorization
```

Lazy wrapper type for the transpose of the underlying `Factorization` object. Usually, the `TransposeFactorization` constructor should not be called directly, use `transpose(:: Factorization)` instead.

`Base.adjoint` – Function.

```
A'
adjoint(A)
```

Lazy adjoint (conjugate transposition). Note that `adjoint` is applied recursively to elements.

For number types, `adjoint` returns the complex conjugate, and therefore it is equivalent to the identity function for real numbers.

This operation is intended for linear algebra usage - for general data manipulation see `permutedims`.

Examples

```
julia> A = [3+2im 9+2im; 0  0]
2×2 Matrix{Complex{Int64}}:
 3+2im  9+2im
 0+0im  0+0im

julia> B = A' # equivalently adjoint(A)
2×2 adjoint(::Matrix{Complex{Int64}}) with eltype Complex{Int64}:
 3-2im  0+0im
 9-2im  0+0im

julia> B isa Adjoint
true

julia> adjoint(B) === A # the adjoint of an adjoint unwraps the parent
true

julia> Adjoint(B) # however, the constructor always wraps its argument
2×2 adjoint(adjoint(::Matrix{Complex{Int64}})) with eltype Complex{Int64}:
 3+2im  9+2im
 0+0im  0+0im

julia> B[1,2] = 4 + 5im; # modifying B will modify A automatically

julia> A
2×2 Matrix{Complex{Int64}}:
 3+2im  9+2im
 4-5im  0+0im
```

For real matrices, the `adjoint` operation is equivalent to a `transpose`.

```
julia> A = reshape([x for x in 1:4], 2, 2)
2×2 Matrix{Int64}:
 1  3
 2  4
```

```
julia> A'
2×2 adjoint(::Matrix{Int64}) with eltype Int64:
 1  2
 3  4

julia> adjoint(A) == transpose(A)
true
```

The adjoint of an `AbstractVector` is a row-vector:

```
julia> x = [3, 4im]
2-element Vector{Complex{Int64}}:
 3 + 0im
 0 + 4im

julia> x'
1×2 adjoint(::Vector{Complex{Int64}}) with eltype Complex{Int64}:
 3+0im  0-4im

julia> x'x # compute the dot product, equivalently x' * x
25 + 0im
```

For a matrix of matrices, the individual blocks are recursively operated on:

```
julia> A = reshape([x + im*x for x in 1:4], 2, 2)
2×2 Matrix{Complex{Int64}}:
 1+1im  3+3im
 2+2im  4+4im

julia> C = reshape([A, 2A, 3A, 4A], 2, 2)
2×2 Matrix{Matrix{Complex{Int64}}}:
 [1+1im 3+3im; 2+2im 4+4im]  [3+3im 9+9im; 6+6im 12+12im]
 [2+2im 6+6im; 4+4im 8+8im]  [4+4im 12+12im; 8+8im 16+16im]

julia> C'
2×2 adjoint(::Matrix{Matrix{Complex{Int64}}}) with eltype Adjoint{Complex{Int64},
↪  Matrix{Complex{Int64}}}:
 [1-1im 2-2im; 3-3im 4-4im]    [2-2im 4-4im; 6-6im 8-8im]
 [3-3im 6-6im; 9-9im 12-12im]  [4-4im 8-8im; 12-12im 16-16im]
```

```
adjoint(F::Factorization)
```

Lazy adjoint of the factorization F. By default, returns an `AdjointFactorization` wrapper.

`LinearAlgebra.adjoint!` – Function.

```
adjoint!(dest,src)
```

Conjugate transpose array `src` and store the result in the preallocated array `dest`, which should have a size corresponding to `(size(src,2),size(src,1))`. No in-place transposition is supported and unexpected results will happen if `src` and `dest` have overlapping memory regions.

Examples

```
julia> A = [3+2im 9+2im; 8+7im  4+6im]
2×2 Matrix{Complex{Int64}}:
 3+2im  9+2im
 8+7im  4+6im

julia> B = zeros(Complex{Int64}, 2, 2)
2×2 Matrix{Complex{Int64}}:
 0+0im  0+0im
 0+0im  0+0im

julia> adjoint!(B, A);

julia> B
2×2 Matrix{Complex{Int64}}:
 3-2im  8-7im
 9-2im  4-6im

julia> A
2×2 Matrix{Complex{Int64}}:
 3+2im  9+2im
 8+7im  4+6im
```

```
adjoint!(X::AbstractSparseMatrixCSC{Tv,Ti}, A::AbstractSparseMatrixCSC{Tv,Ti}) where {Tv,Ti}
```

Transpose the matrix A and stores the adjoint of the elements in the matrix X. `size(X)` must be equal to `size(transpose(A))`. No additional memory is allocated other than resizing the rowval and nzval of X, if needed.

See `halfperm!`

`LinearAlgebra.Adjoint` – Type.

```
Adjoint
```

Lazy wrapper type for an adjoint view of the underlying linear algebra object, usually an `AbstractVector`/`AbstractMatrix`. Usually, the `Adjoint` constructor should not be called directly, use `adjoint` instead. To materialize the view use `copy`.

This type is intended for linear algebra usage - for general data manipulation see `permutedims`.

Examples

```
julia> A = [3+2im 9+2im; 0 0]
2×2 Matrix{Complex{Int64}}:
 3+2im  9+2im
 0+0im  0+0im

julia> Adjoint(A)
2×2 adjoint(::Matrix{Complex{Int64}}) with eltype Complex{Int64}:
 3-2im  0+0im
 9-2im  0+0im
```

`LinearAlgebra.AdjointFactorization` – Type.

```
AdjointFactorization
```

Lazy wrapper type for the adjoint of the underlying `Factorization` object. Usually, the `AdjointFactorization` constructor should not be called directly, use `adjoint(:: Factorization)` instead.

`Base.copy` – Method.

```
copy(A::Transpose)
copy(A::Adjoint)
```

Eagerly evaluate the lazy matrix transpose/adjoint. Note that the transposition is applied recursively to elements.

This operation is intended for linear algebra usage - for general data manipulation see `permutedims`, which is non-recursive.

Examples

```
julia> A = [1 2im; -3im 4]
2×2 Matrix{Complex{Int64}}:
 1+0im  0+2im
 0-3im  4+0im

julia> T = transpose(A)
2×2 transpose(::Matrix{Complex{Int64}}) with eltype Complex{Int64}:
 1+0im  0-3im
 0+2im  4+0im

julia> copy(T)
2×2 Matrix{Complex{Int64}}:
 1+0im  0-3im
 0+2im  4+0im
```

`LinearAlgebra.stride1` – Function.

```
stride1(A) -> Int
```

Return the distance between successive array elements in dimension 1 in units of element size.

Examples

```
julia> A = [1,2,3,4]
4-element Vector{Int64}:
 1
 2
 3
 4

julia> LinearAlgebra.stride1(A)
1

julia> B = view(A, 2:2:4)
2-element view(::Vector{Int64}, 2:2:4) with eltype Int64:
 2
 4

julia> LinearAlgebra.stride1(B)
2
```

`LinearAlgebra.checksquare` – Function.

```
LinearAlgebra.checksquare(A)
```

Check that a matrix is square, then return its common dimension. For multiple arguments, return a vector.

Examples

```
julia> A = fill(1, (4,4)); B = fill(1, (5,5));

julia> LinearAlgebra.checksquare(A, B)
2-element Vector{Int64}:
 4
 5
```

`LinearAlgebra.peakflops` – Function.

```
LinearAlgebra.peakflops(n::Integer=4096; eltype::DataType=Float64, ntrials::Integer=3,
↪  parallel::Bool=false)
```

`peakflops` computes the peak flop rate of the computer by using double precision `gemm!`. By default, if no arguments are specified, it multiplies two `Float64` matrices of size `n x n`, where `n = 4096`. If the underlying BLAS is using multiple threads, higher flop rates are realized. The number of BLAS threads can be set with `BLAS.set_num_threads(n)`.

If the keyword argument `eltype` is provided, `peakflops` will construct matrices with elements of type `eltype` for calculating the peak flop rate.

By default, `peakflops` will use the best timing from 3 trials. If the `ntrials` keyword argument is provided, `peakflops` will use those many trials for picking the best timing.

If the keyword argument `parallel` is set to `true`, `peakflops` is run in parallel on all the worker processors. The flop rate of the entire parallel computer is returned. When running in parallel, only 1 BLAS thread is used. The argument `n` still refers to the size of the problem that is solved on each processor.

Julia 1.1

This function requires at least Julia 1.1. In Julia 1.0 it is available from the standard library `InteractiveUtils`.

`LinearAlgebra.hermitianpart` – Function.

```
hermitianpart(A::AbstractMatrix, uplo::Symbol=:U) -> Hermitian
```

Return the Hermitian part of the square matrix `A`, defined as `(A + A') / 2`, as a `Hermitian` matrix. For real matrices `A`, this is also known as the symmetric part of `A`; it is also sometimes called the "operator real part". The optional argument `uplo` controls the corresponding argument of the `Hermitian` view. For real matrices, the latter is equivalent to a `Symmetric` view.

See also `hermitianpart!` for the corresponding in-place operation.

Julia 1.10

This function requires Julia 1.10 or later.

`LinearAlgebra.hermitianpart!` – Function.

```
hermitianpart!(A::AbstractMatrix, uplo::Symbol=:U) -> Hermitian
```

Overwrite the square matrix `A` in-place with its Hermitian part `(A + A') / 2`, and return `Hermitian(A, uplo)`. For real matrices `A`, this is also known as the symmetric part of `A`.

See also `hermitianpart` for the corresponding out-of-place operation.

Julia 1.10

This function requires Julia 1.10 or later.

`LinearAlgebra.copy_adjoint!` – Function.

```
copy_adjoint!(B::AbstractVecOrMat, ir_dest::AbstractRange{Int}, jr_dest::AbstractRange{Int},
              A::AbstractVecOrMat, ir_src::AbstractRange{Int}, jr_src::AbstractRange{Int})
              ↪  -> B
```

Efficiently copy elements of matrix `A` to `B` with adjunction as follows:

```
B[ir_dest, jr_dest] = adjoint(A)[jr_src, ir_src]
```

The elements `B[ir_dest, jr_dest]` are overwritten. Furthermore, the index range parameters must satisfy `length(ir_dest) == length(jr_src)` and `length(jr_dest) == length(ir_src)`.

`LinearAlgebra.copy_transpose!` – Function.

```
copy_transpose!(B::AbstractVecOrMat, ir_dest::AbstractRange{Int}, jr_dest::AbstractRange{Int},
              A::AbstractVecOrMat, ir_src::AbstractRange{Int}, jr_src::AbstractRange{Int})
              ↪  -> B
```

Efficiently copy elements of matrix `A` to `B` with transposition as follows:

```
B[ir_dest, jr_dest] = transpose(A)[jr_src, ir_src]
```

The elements `B[ir_dest, jr_dest]` are overwritten. Furthermore, the index range parameters must satisfy `length(ir_dest) == length(jr_src)` and `length(jr_dest) == length(ir_src)`.

```
copy_transpose!(B::AbstractMatrix, ir_dest::AbstractUnitRange, jr_dest::AbstractUnitRange,
              tM::AbstractChar,
              M::AbstractVecOrMat, ir_src::AbstractUnitRange, jr_src::AbstractUnitRange) ->
              ↪  B
```

Efficiently copy elements of matrix `M` to `B` conditioned on the character parameter `tM` as follows:

tM	Destination	Source
`'N'`	`B[ir_dest, jr_dest]`	`transpose(M)[jr_src, ir_src]`
`'T'`	`B[ir_dest, jr_dest]`	`M[jr_src, ir_src]`
`'C'`	`B[ir_dest, jr_dest]`	`conj(M)[jr_src, ir_src]`

The elements `B[ir_dest, jr_dest]` are overwritten. Furthermore, the index range parameters must satisfy `length(ir_dest) == length(jr_src)` and `length(jr_dest) == length(ir_src)`.

See also `copyto!` and `copy_adjoint!`.

`LinearAlgebra.uplo` – Function.

```
LinearAlgebra.uplo(S::Union{Symmetric, Hermitian})::Symbol
```

Return a `Symbol` corresponding to the stored triangular half (`:U` or `:L`) in the matrix `S`, that is, the elements are common between `S` and `parent(S)` for that triangular half.

Example

```
julia> S = Symmetric([1 2; 3 4], :U)
2×2 Symmetric{Int64, Matrix{Int64}}:
 1  2
 2  4

julia> LinearAlgebra.uplo(S)
:U

julia> H = Hermitian([1 2; 3 4], :L)
2×2 Hermitian{Int64, Matrix{Int64}}:
 1  3
 3  4

julia> LinearAlgebra.uplo(H)
:L
```

```
LinearAlgebra.uplo(S::Bidiagonal)::Symbol
```

Return a `Symbol` corresponding to whether the upper (`:U`) or lower (`:L`) off-diagonal band is stored.

83.6 Low-level matrix operations

In many cases there are in-place versions of matrix operations that allow you to supply a pre-allocated output vector or matrix. This is useful when optimizing critical code in order to avoid the overhead of repeated allocations. These in-place operations are suffixed with `!` below (e.g. `mul!`) according to the usual Julia convention.

`LinearAlgebra.mul!` – Function.

```
mul!(Y, A, B) -> Y
```

Calculates the matrix-matrix or matrix-vector product AB and stores the result in `Y`, overwriting the existing value of `Y`. Note that `Y` must not be aliased with either `A` or `B`.

Examples

```
julia> A = [1.0 2.0; 3.0 4.0]; B = [1.0 1.0; 1.0 1.0]; Y = similar(B);

julia> mul!(Y, A, B) === Y
true

julia> Y
2×2 Matrix{Float64}:
 3.0  3.0
```

```
 7.0  7.0

julia> Y == A * B
true
```

Implementation

For custom matrix and vector types, it is recommended to implement 5-argument `mul!` rather than implementing 3-argument `mul!` directly if possible.

```
mul!(C, A, B, α, β) -> C
```

Combined inplace matrix-matrix or matrix-vector multiply-add $AB + C$. The result is stored in `C` by overwriting it. Note that `C` must not be aliased with either `A` or `B`.

Julia 1.3

Five-argument `mul!` requires at least Julia 1.3.

Examples

```
julia> A = [1.0 2.0; 3.0 4.0]; B = [1.0 1.0; 1.0 1.0]; C = [1.0 2.0; 3.0 4.0];

julia> α, β = 100.0, 10.0;

julia> mul!(C, A, B, α, β) === C
true

julia> C
2×2 Matrix{Float64}:
 310.0  320.0
 730.0  740.0

julia> C_original = [1.0 2.0; 3.0 4.0]; # A copy of the original value of C

julia> C == A * B * α + C_original * β
true
```

```
mul!(c::AbstractVector, tA, A::AbstractVecOrMat, b::AbstractVector, α::Number, β::Number)
```

Calculates the combined matrix-vector multiply-add $Ab+c$ (`tA == 'N'`), $Ab+c$ (`tA == 'T'`), or $A'b+c$ (`tA == 'C'`). The result is stored in `c` by overwriting it. Note that `c` must not be aliased with either `A` or `b`.

This is an abstraction layer below 5-arg `mul!` used to dispatch on storage types. Packages that provide their own storage type are advised to overload this method signature instead of 5-arg `mul!`.

Julia 1.13

This method requires at least Julia 1.13 and supersedes the non-public `generic_matvecmul!` method with the same signature.

```
mul!(C::AbstractVecOrMat, tA, tB, A::AbstractVecOrMat, B::AbstractVecOrMat, α::Number,
↪  β::Number)
```

Calculates the combined matrix-matrix multiply-add $AB + C$ (`tA == 'N'`), $AB + C$ (`tA == 'T'`), or $A'B+C$ (`tA == 'C'`), with potential matrix transpositions of `B` corresponding to `tB`. The result is stored in `C` by overwriting it. Note that `C` must not be aliased with either `A` or `B`.

This is an abstraction layer below 5-arg `mul!`. Packages that provide their own storage type are advised to overload this method instead of 5-arg `mul!`.

Julia 1.13

This method requires at least Julia 1.13 and supersedes the non-public `generic_matmatmul!` method with the same signature.

`LinearAlgebra.lmul!` – Function.

```
lmul!(a::Number, B::AbstractArray)
```

Scale an array `B` by a scalar `a` overwriting `B` in-place. Use `rmul!` to multiply scalar from right. The scaling operation respects the semantics of the multiplication `*` between `a` and an element of `B`. In particular, this also applies to multiplication involving non-finite numbers such as `NaN` and `±Inf`.

Julia 1.1

Prior to Julia 1.1, `NaN` and `±Inf` entries in `B` were treated inconsistently.

Examples

```
julia> B = [1 2; 3 4]
2×2 Matrix{Int64}:
 1  2
 3  4

julia> lmul!(2, B)
2×2 Matrix{Int64}:
 2  4
 6  8

julia> lmul!(0.0, [Inf])
1-element Vector{Float64}:
 NaN
```

```
lmul!(A, B)
```

Calculate the matrix-matrix product AB, overwriting `B`, and return the result. Here, `A` must be of special matrix type, like, e.g., `Diagonal`, `UpperTriangular` or `LowerTriangular`, or of some orthogonal type, see `QR`.

Examples

```
julia> B = [0 1; 1 0];
```

```
julia> A = UpperTriangular([1 2; 0 3]);

julia> lmul!(A, B);

julia> B
2×2 Matrix{Int64}:
 2  1
 3  0

julia> B = [1.0 2.0; 3.0 4.0];

julia> F = qr([0 1; -1 0]);

julia> lmul!(F.Q, B)
2×2 Matrix{Float64}:
 3.0  4.0
 1.0  2.0
```

`LinearAlgebra.rmul!` – Function.

```
rmul!(A::AbstractArray, b::Number)
```

Scale an array A by a scalar b overwriting A in-place. Use `lmul!` to multiply scalar from left. The scaling operation respects the semantics of the multiplication `*` between an element of A and b. In particular, this also applies to multiplication involving non-finite numbers such as `NaN` and `±Inf`.

Julia 1.1

Prior to Julia 1.1, `NaN` and `±Inf` entries in A were treated inconsistently.

Examples

```
julia> A = [1 2; 3 4]
2×2 Matrix{Int64}:
 1  2
 3  4

julia> rmul!(A, 2)
2×2 Matrix{Int64}:
 2  4
 6  8

julia> rmul!([NaN], 0.0)
1-element Vector{Float64}:
 NaN
```

```
rmul!(A, B)
```

Calculate the matrix-matrix product AB, overwriting A, and return the result. Here, B must be of special matrix type, like, e.g., `Diagonal`, `UpperTriangular` or `LowerTriangular`, or of some orthogonal type, see `QR`.

Examples

```
julia> A = [0 1; 1 0];

julia> B = UpperTriangular([1 2; 0 3]);

julia> rmul!(A, B);

julia> A
2×2 Matrix{Int64}:
 0  3
 1  2

julia> A = [1.0 2.0; 3.0 4.0];

julia> F = qr([0 1; -1 0]);

julia> rmul!(A, F.Q)
2×2 Matrix{Float64}:
 2.0  1.0
 4.0  3.0
```

`LinearAlgebra.ldiv!` – Function.

```
ldiv!(Y, A, B) -> Y
```

Compute `A \ B` in-place and store the result in `Y`, returning the result.

The argument `A` should *not* be a matrix. Rather, instead of matrices it should be a factorization object (e.g. produced by `factorize` or `cholesky`). The reason for this is that factorization itself is both expensive and typically allocates memory (although it can also be done in-place via, e.g., `lu!`), and performance-critical situations requiring `ldiv!` usually also require fine-grained control over the factorization of `A`.

Note

Certain structured matrix types, such as `Diagonal` and `UpperTriangular`, are permitted, as these are already in a factorized form

Examples

```
julia> A = [1 2.2 4; 3.1 0.2 3; 4 1 2];

julia> B = [1, 2.5, 3];

julia> Y = similar(B); # use similar since there is no need to read from it

julia> ldiv!(Y, qr(A), B); # you may also try qr!(A) to further reduce allocation

julia> Y ≈ A \ B
true
```

```
ldiv!(A, B)
```

Compute `A \ B` in-place and overwriting `B` to store the result.

The argument `A` should *not* be a matrix. Rather, instead of matrices it should be a factorization object (e.g. produced by `factorize` or `cholesky`). The reason for this is that factorization itself is both expensive and typically allocates memory (although it can also be done in-place via, e.g., `lu!`), and performance-critical situations requiring `ldiv!` usually also require fine-grained control over the factorization of `A`.

Note

Certain structured matrix types, such as `Diagonal` and `UpperTriangular`, are permitted, as these are already in a factorized form

Examples

```
julia> A = [1 2.2 4; 3.1 0.2 3; 4 1 2];

julia> B = [1, 2.5, 3];

julia> B0 = copy(B); # a backup copy to facilitate testing

julia> ldiv!(lu(A), B); # you may also try lu!(A) to further reduce allocation

julia> B ≈ A \ B0
true
```

```
ldiv!(a::Number, B::AbstractArray)
```

Divide each entry in an array `B` by a scalar `a` overwriting `B` in-place. Use `rdiv!` to divide scalar from right.

Examples

```
julia> B = [1.0 2.0; 3.0 4.0]
2×2 Matrix{Float64}:
 1.0  2.0
 3.0  4.0

julia> ldiv!(2.0, B)
2×2 Matrix{Float64}:
 0.5  1.0
 1.5  2.0
```

```
ldiv!(A::Tridiagonal, B::AbstractVecOrMat) -> B
```

Compute `A \ B` in-place by Gaussian elimination with partial pivoting and store the result in `B`, returning the result. In the process, the diagonals of `A` are overwritten as well.

Julia 1.11

`ldiv!` for `Tridiagonal` left-hand sides requires at least Julia 1.11.

```
ldiv!(F::SVD, B; atol::Real=0, rtol::Real=atol>0 ? 0 : n*ϵ)
```

Given the SVD F of an $m \times n$ matrix, multiply the first m rows of B in-place by the Moore-Penrose pseudoinverse, storing the result in the first n rows of B, returning B. This is equivalent to a least-squares solution (for $m \geq n$) or a minimum-norm solution (for $m \leq n$).

Similar to the pinv function, the solution can be regularized by truncating the SVD, dropping any singular values less than `max(atol, rtol*σ₁)` where σ_1 is the largest singular value. The default relative tolerance is `n*ϵ`, where n is the size of the smallest dimension of M, and ϵ is the eps of the element type of M.

Julia 1.13

The atol and rtol arguments require Julia 1.13 or later.

`LinearAlgebra.rdiv!` – Function.

```
rdiv!(A, B)
```

Compute `A / B` in-place and overwriting A to store the result.

The argument B should *not* be a matrix. Rather, instead of matrices it should be a factorization object (e.g. produced by factorize or cholesky). The reason for this is that factorization itself is both expensive and typically allocates memory (although it can also be done in-place via, e.g., lu!), and performance-critical situations requiring `rdiv!` usually also require fine-grained control over the factorization of B.

Note

Certain structured matrix types, such as Diagonal and UpperTriangular, are permitted, as these are already in a factorized form

```
rdiv!(A::AbstractArray, b::Number)
```

Divide each entry in an array A by a scalar b overwriting A in-place. Use ldiv! to divide scalar from left.

Examples

```
julia> A = [1.0 2.0; 3.0 4.0]
2×2 Matrix{Float64}:
 1.0  2.0
 3.0  4.0

julia> rdiv!(A, 2.0)
2×2 Matrix{Float64}:
 0.5  1.0
 1.5  2.0
```

83.7 BLAS functions

In Julia (as in much of scientific computation), dense linear-algebra operations are based on the LAPACK library, which in turn is built on top of basic linear-algebra building-blocks known as the BLAS. There are

highly optimized implementations of BLAS available for every computer architecture, and sometimes in high-performance linear algebra routines it is useful to call the BLAS functions directly.

`LinearAlgebra.BLAS` provides wrappers for some of the BLAS functions. Those BLAS functions that overwrite one of the input arrays have names ending in `'!'`. Usually, a BLAS function has four methods defined, for `Float32`, `Float64`, `ComplexF32`, and `ComplexF64` arrays.

BLAS character arguments

Many BLAS functions accept arguments that determine whether to transpose an argument (`trans`), which triangle of a matrix to reference (`uplo` or `ul`), whether the diagonal of a triangular matrix can be assumed to be all ones (`dA`) or which side of a matrix multiplication the input argument belongs on (`side`). The possibilities are:

Multiplication order

side	Meaning
`'L'`	The argument goes on the *left* side of a matrix-matrix operation.
`'R'`	The argument goes on the *right* side of a matrix-matrix operation.

Triangle referencing

uplo/ul	Meaning
`'U'`	Only the *upper* triangle of the matrix will be used.
`'L'`	Only the *lower* triangle of the matrix will be used.

Transposition operation

trans/tX	Meaning
`'N'`	The input matrix X is not transposed or conjugated.
`'T'`	The input matrix X will be transposed.
`'C'`	The input matrix X will be conjugated and transposed.

Unit diagonal

diag/dX	Meaning
`'N'`	The diagonal values of the matrix X will be read.
`'U'`	The diagonal of the matrix X is assumed to be all ones.

`LinearAlgebra.BLAS` – Module.

Interface to BLAS subroutines.

`LinearAlgebra.BLAS.set_num_threads` – Function.

```
set_num_threads(n::Integer)
set_num_threads(::Nothing)
```

Set the number of threads the BLAS library should use equal to `n::Integer`.

Also accepts `nothing`, in which case julia tries to guess the default number of threads. Passing `nothing` is discouraged and mainly exists for historical reasons.

Note

Some BLAS libraries, such as Apple Accelerate, cannot be configured to use a fixed number of threads. For these backends, `set_num_threads()` is a no-op. See also `get_num_threads`.

`LinearAlgebra.BLAS.get_num_threads` – Function.

```
get_num_threads()
```

Get the number of threads the BLAS library is using.

Julia 1.6

`get_num_threads` requires at least Julia 1.6.

Note

Some BLAS libraries, such as Apple Accelerate, cannot be configured to use a fixed number of threads. For these backends, `get_num_threads()` always returns 1. See also `set_num_threads`.

BLAS functions can be divided into three groups, also called three levels, depending on when they were first proposed, the type of input parameters, and the complexity of the operation.

Level 1 BLAS functions

The level 1 BLAS functions were first proposed in (Lawson, 1979) and define operations between scalars and vectors.

`LinearAlgebra.BLAS.rot!` – Function.

```
rot!(n, X, incx, Y, incy, c, s)
```

Overwrite X with `c*X + s*Y` and Y with `-conj(s)*X + c*Y` for the first n elements of array X with stride `incx` and first n elements of array Y with stride `incy`. Returns X and Y.

Julia 1.5

`rot!` requires at least Julia 1.5.

`LinearAlgebra.BLAS.scal!` – Function.

```
scal!(n, a, X, incx)
scal!(a, X)
```

Overwrite X with a*X for the first n elements of array X with stride `incx`. Returns X.

If n and `incx` are not provided, `length(X)` and `stride(X,1)` are used.

`LinearAlgebra.BLAS.scal` – Function.

```
scal(n, a, X, incx)
scal(a, X)
```

Return X scaled by a for the first n elements of array X with stride `incx`.

If n and `incx` are not provided, `length(X)` and `stride(X,1)` are used.

`LinearAlgebra.BLAS.blascopy!` – Function.

```
blascopy!(n, X, incx, Y, incy)
```

Copy n elements of array X with stride `incx` to array Y with stride `incy`. Returns Y.

`LinearAlgebra.BLAS.dot` – Function.

```
dot(n, X, incx, Y, incy)
```

Dot product of two vectors consisting of n elements of array X with stride `incx` and n elements of array Y with stride `incy`.

Examples

```
julia> BLAS.dot(10, fill(1.0, 10), 1, fill(1.0, 20), 2)
10.0
```

`LinearAlgebra.BLAS.dotu` – Function.

```
dotu(n, X, incx, Y, incy)
```

Dot function for two complex vectors consisting of n elements of array X with stride `incx` and n elements of array Y with stride `incy`.

Examples

```
julia> BLAS.dotu(10, fill(1.0im, 10), 1, fill(1.0+im, 20), 2)
-10.0 + 10.0im
```

`LinearAlgebra.BLAS.dotc` – Function.

```
dotc(n, X, incx, U, incy)
```

Dot function for two complex vectors, consisting of n elements of array X with stride `incx` and n elements of array U with stride `incy`, conjugating the first vector.

Examples

```
julia> BLAS.dotc(10, fill(1.0im, 10), 1, fill(1.0+im, 20), 2)
10.0 - 10.0im
```

`LinearAlgebra.BLAS.nrm2` – Function.

```
nrm2(n, X, incx)
```

2-norm of a vector consisting of n elements of array X with stride `incx`.

Examples

```
julia> BLAS.nrm2(4, fill(1.0, 8), 2)
2.0

julia> BLAS.nrm2(1, fill(1.0, 8), 2)
1.0
```

`LinearAlgebra.BLAS.asum` – Function.

```
asum(n, X, incx)
```

Sum of the magnitudes of the first n elements of array X with stride `incx`.

For a real array, the magnitude is the absolute value. For a complex array, the magnitude is the sum of the absolute value of the real part and the absolute value of the imaginary part.

Examples

```
julia> BLAS.asum(5, fill(1.0im, 10), 2)
5.0

julia> BLAS.asum(2, fill(1.0im, 10), 5)
2.0
```

`LinearAlgebra.BLAS.iamax` – Function.

```
iamax(n, dx, incx)
iamax(dx)
```

Find the index of the element of `dx` with the maximum absolute value. `n` is the length of `dx`, and `incx` is the stride. If `n` and `incx` are not provided, they assume default values of `n=length(dx)` and `incx=stride1(dx)`.

Level 2 BLAS functions

The level 2 BLAS functions were published in (Dongarra, 1988) and define matrix-vector operations.

return a vector

`LinearAlgebra.BLAS.gemv!` – Function.

```
gemv!(tA, alpha, A, x, beta, y)
```

Update the vector y as `alpha*A*x + beta*y` or `alpha*A'x + beta*y` according to `tA`. `alpha` and `beta` are scalars. Return the updated y.

`LinearAlgebra.BLAS.gemv` – Method.

```
gemv(tA, alpha, A, x)
```

Return alpha*A*x or alpha*A'x according to tA. alpha is a scalar.

LinearAlgebra.BLAS.gemv – Method.

```
gemv(tA, A, x)
```

Return A*x or A'x according to tA.

LinearAlgebra.BLAS.gbmv! – Function.

```
gbmv!(trans, m, kl, ku, alpha, A, x, beta, y)
```

Update vector y as alpha*A*x + beta*y or alpha*A'*x + beta*y according to trans. The matrix A is a general band matrix of dimension m by size(A,2) with kl sub-diagonals and ku super-diagonals. alpha and beta are scalars. Return the updated y.

LinearAlgebra.BLAS.gbmv – Function.

```
gbmv(trans, m, kl, ku, alpha, A, x)
```

Return alpha*A*x or alpha*A'*x according to trans. The matrix A is a general band matrix of dimension m by size(A,2) with kl sub-diagonals and ku super-diagonals, and alpha is a scalar.

LinearAlgebra.BLAS.hemv! – Function.

```
hemv!(ul, alpha, A, x, beta, y)
```

Update the vector y as alpha*A*x + beta*y. A is assumed to be Hermitian. Only the ul triangle of A is used. alpha and beta are scalars. Return the updated y.

LinearAlgebra.BLAS.hemv – Method.

```
hemv(ul, alpha, A, x)
```

Return alpha*A*x. A is assumed to be Hermitian. Only the ul triangle of A is used. alpha is a scalar.

LinearAlgebra.BLAS.hemv – Method.

```
hemv(ul, A, x)
```

Return A*x. A is assumed to be Hermitian. Only the ul triangle of A is used.

LinearAlgebra.BLAS.hpmv! – Function.

```
hpmv!(uplo, α, AP, x, β, y)
```

Update vector y as α*A*x + β*y, where A is a Hermitian matrix provided in packed format AP.

With uplo = 'U', the array AP must contain the upper triangular part of the Hermitian matrix packed sequentially, column by column, so that AP[1] contains A[1, 1], AP[2] and AP[3] contain A[1, 2] and A[2, 2] respectively, and so on.

With uplo = 'L', the array AP must contain the lower triangular part of the Hermitian matrix packed sequentially, column by column, so that AP[1] contains A[1, 1], AP[2] and AP[3] contain A[2, 1] and A[3, 1] respectively, and so on.

The scalar inputs α and β must be complex or real numbers.

The array inputs x, y and AP must all be of `ComplexF32` or `ComplexF64` type.

Return the updated y.

Julia 1.5

`hpmv!` requires at least Julia 1.5.

`LinearAlgebra.BLAS.symv!` – Function.

```
symv!(ul, alpha, A, x, beta, y)
```

Update the vector y as `alpha*A*x + beta*y`. A is assumed to be symmetric. Only the `ul` triangle of A is used. `alpha` and `beta` are scalars. Return the updated y.

`LinearAlgebra.BLAS.symv` – Method.

```
symv(ul, alpha, A, x)
```

Return `alpha*A*x`. A is assumed to be symmetric. Only the `ul` triangle of A is used. `alpha` is a scalar.

`LinearAlgebra.BLAS.symv` – Method.

```
symv(ul, A, x)
```

Return `A*x`. A is assumed to be symmetric. Only the `ul` triangle of A is used.

`LinearAlgebra.BLAS.sbmv!` – Function.

```
sbmv!(uplo, k, alpha, A, x, beta, y)
```

Update vector y as `alpha*A*x + beta*y` where A is a symmetric band matrix of order `size(A,2)` with k super-diagonals stored in the argument A. The storage layout for A is described the reference BLAS module, level-2 BLAS at https://www.netlib.org/lapack/explore-html/. Only the `uplo` triangle of A is used.

Return the updated y.

`LinearAlgebra.BLAS.sbmv` – Method.

```
sbmv(uplo, k, alpha, A, x)
```

Return `alpha*A*x` where A is a symmetric band matrix of order `size(A,2)` with k super-diagonals stored in the argument A. Only the `uplo` triangle of A is used.

`LinearAlgebra.BLAS.sbmv` – Method.

```
sbmv(uplo, k, A, x)
```

Return `A*x` where A is a symmetric band matrix of order `size(A,2)` with k super-diagonals stored in the argument A. Only the `uplo` triangle of A is used.

`LinearAlgebra.BLAS.spmv!` – Function.

```
spmv!(uplo, α, AP, x, β, y)
```

Update vector y as α*A*x + β*y, where A is a symmetric matrix provided in packed format AP.

With uplo = 'U', the array AP must contain the upper triangular part of the symmetric matrix packed sequentially, column by column, so that AP[1] contains A[1, 1], AP[2] and AP[3] contain A[1, 2] and A[2, 2] respectively, and so on.

With uplo = 'L', the array AP must contain the lower triangular part of the symmetric matrix packed sequentially, column by column, so that AP[1] contains A[1, 1], AP[2] and AP[3] contain A[2, 1] and A[3, 1] respectively, and so on.

The scalar inputs α and β must be real.

The array inputs x, y and AP must all be of Float32 or Float64 type.

Return the updated y.

Julia 1.5

spmv! requires at least Julia 1.5.

LinearAlgebra.BLAS.trmv! – Function.

```
trmv!(ul, tA, dA, A, b)
```

Return op(A)*b, where op is determined by tA. Only the ul triangle of A is used. dA determines if the diagonal values are read or are assumed to be all ones. The multiplication occurs in-place on b.

LinearAlgebra.BLAS.trmv – Function.

```
trmv(ul, tA, dA, A, b)
```

Return op(A)*b, where op is determined by tA. Only the ul triangle of A is used. dA determines if the diagonal values are read or are assumed to be all ones.

LinearAlgebra.BLAS.trsv! – Function.

```
trsv!(ul, tA, dA, A, b)
```

Overwrite b with the solution to A*x = b or one of the other two variants determined by tA and ul. dA determines if the diagonal values are read or are assumed to be all ones. Return the updated b.

LinearAlgebra.BLAS.trsv – Function.

```
trsv(ul, tA, dA, A, b)
```

Return the solution to A*x = b or one of the other two variants determined by tA and ul. dA determines if the diagonal values are read or are assumed to be all ones.

return a matrix

LinearAlgebra.BLAS.ger! – Function.

```
ger!(alpha, x, y, A)
```

Rank-1 update of the matrix A with vectors x and y as `alpha*x*y' + A`.

`LinearAlgebra.BLAS.geru!` – Function.

```
geru!(alpha, x, y, A)
```

Rank-1 update of the matrix A with vectors x and y as `alpha*x*transpose(y) + A`.

`LinearAlgebra.BLAS.her!` – Function.

```
her!(uplo, alpha, x, A)
```

Methods for complex arrays only. Rank-1 update of the Hermitian matrix A with vector x as `alpha*x*x' + A`. `uplo` controls which triangle of A is updated. Returns A.

`LinearAlgebra.BLAS.syr!` – Function.

```
syr!(uplo, alpha, x, A)
```

Rank-1 update of the symmetric matrix A with vector x as `alpha*x*transpose(x) + A`. `uplo` controls which triangle of A is updated. Returns A.

`LinearAlgebra.BLAS.spr!` – Function.

```
spr!(uplo, α, x, AP)
```

Update matrix A as A+α*x*x', where A is a symmetric matrix provided in packed format AP and x is a vector.

With `uplo = 'U'`, the array AP must contain the upper triangular part of the symmetric matrix packed sequentially, column by column, so that `AP[1]` contains `A[1, 1]`, `AP[2]` and `AP[3]` contain `A[1, 2]` and `A[2, 2]` respectively, and so on.

With `uplo = 'L'`, the array AP must contain the lower triangular part of the symmetric matrix packed sequentially, column by column, so that `AP[1]` contains `A[1, 1]`, `AP[2]` and `AP[3]` contain `A[2, 1]` and `A[3, 1]` respectively, and so on.

The scalar input α must be real.

The array inputs x and AP must all be of `Float32` or `Float64` type. Return the updated AP.

Julia 1.8

`spr!` requires at least Julia 1.8.

Level 3 BLAS functions

The level 3 BLAS functions were published in (Dongarra, 1990) and define matrix-matrix operations.

`LinearAlgebra.BLAS.gemmt!` – Function.

```
gemmt!(uplo, tA, tB, alpha, A, B, beta, C)
```

Update the lower or upper triangular part specified by `uplo` of `C` as `alpha*A*B + beta*C` or the other variants according to `tA` and `tB`. Return the updated `C`.

> **Julia 1.11**
>
> `gemmt!` requires at least Julia 1.11.

`LinearAlgebra.BLAS.gemmt` – Method.

```
gemmt(uplo, tA, tB, alpha, A, B)
```

Return the lower or upper triangular part specified by `uplo` of `A*B` or the other three variants according to `tA` and `tB`.

> **Julia 1.11**
>
> `gemmt` requires at least Julia 1.11.

`LinearAlgebra.BLAS.gemmt` – Method.

```
gemmt(uplo, tA, tB, A, B)
```

Return the lower or upper triangular part specified by `uplo` of `A*B` or the other three variants according to `tA` and `tB`.

> **Julia 1.11**
>
> `gemmt` requires at least Julia 1.11.

`LinearAlgebra.BLAS.gemm!` – Function.

```
gemm!(tA, tB, alpha, A, B, beta, C)
```

Update `C` as `alpha*A*B + beta*C` or the other three variants according to `tA` and `tB`. Return the updated `C`.

`LinearAlgebra.BLAS.gemm` – Method.

```
gemm(tA, tB, alpha, A, B)
```

Return `alpha*A*B` or the other three variants according to `tA` and `tB`.

`LinearAlgebra.BLAS.gemm` – Method.

```
gemm(tA, tB, A, B)
```

Return `A*B` or the other three variants according to `tA` and `tB`.

`LinearAlgebra.BLAS.symm!` – Function.

```
symm!(side, ul, alpha, A, B, beta, C)
```

Update `C` as `alpha*A*B + beta*C` or `alpha*B*A + beta*C` according to `side`. `A` is assumed to be symmetric. Only the `ul` triangle of `A` is used. Return the updated `C`.

LinearAlgebra.BLAS.symm – Method.

```
symm(side, ul, alpha, A, B)
```

Return alpha*A*B or alpha*B*A according to side. A is assumed to be symmetric. Only the ul triangle of A is used.

LinearAlgebra.BLAS.symm – Method.

```
symm(side, ul, A, B)
```

Return A*B or B*A according to side. A is assumed to be symmetric. Only the ul triangle of A is used.

LinearAlgebra.BLAS.hemm! – Function.

```
hemm!(side, ul, alpha, A, B, beta, C)
```

Update C as alpha*A*B + beta*C or alpha*B*A + beta*C according to side. A is assumed to be Hermitian. Only the ul triangle of A is used. Return the updated C.

LinearAlgebra.BLAS.hemm – Method.

```
hemm(side, ul, alpha, A, B)
```

Return alpha*A*B or alpha*B*A according to side. A is assumed to be Hermitian. Only the ul triangle of A is used.

LinearAlgebra.BLAS.hemm – Method.

```
hemm(side, ul, A, B)
```

Return A*B or B*A according to side. A is assumed to be Hermitian. Only the ul triangle of A is used.

LinearAlgebra.BLAS.syrk! – Function.

```
syrk!(uplo, trans, alpha, A, beta, C)
```

Rank-k update of the symmetric matrix C as alpha*A*transpose(A) + beta*C or alpha*transpose(A)*A + beta*C according to trans. Only the uplo triangle of C is used. Return C.

LinearAlgebra.BLAS.syrk – Function.

```
syrk(uplo, trans, alpha, A)
```

Return either the upper triangle or the lower triangle of A, according to uplo, of alpha*A*transpose(A) or alpha*transpose(A)*A, according to trans.

LinearAlgebra.BLAS.herk! – Function.

```
herk!(uplo, trans, alpha, A, beta, C)
```

Methods for complex arrays only. Rank-k update of the Hermitian matrix C as alpha*A*A' + beta*C or alpha*A'*A + beta*C according to trans. Only the uplo triangle of C is updated. Returns C.

LinearAlgebra.BLAS.herk – Function.

```
herk(uplo, trans, alpha, A)
```

Methods for complex arrays only. Returns the uplo triangle of alpha*A*A' or alpha*A'*A, according to trans.

LinearAlgebra.BLAS.syr2k! – Function.

```
syr2k!(uplo, trans, alpha, A, B, beta, C)
```

Rank-2k update of the symmetric matrix C as alpha*A*transpose(B) + alpha*B*transpose(A) + beta*C or alpha*transpose(A)*B + alpha*transpose(B)*A + beta*C according to trans. Only the uplo triangle of C is used. Returns C.

LinearAlgebra.BLAS.syr2k – Function.

```
syr2k(uplo, trans, alpha, A, B)
```

Returns the uplo triangle of alpha*A*transpose(B) + alpha*B*transpose(A) or alpha*transpose(A)*B + alpha*transpose(B)*A, according to trans.

```
syr2k(uplo, trans, A, B)
```

Return the uplo triangle of A*transpose(B) + B*transpose(A) or transpose(A)*B + transpose(B)*A, according to trans.

LinearAlgebra.BLAS.her2k! – Function.

```
her2k!(uplo, trans, alpha, A, B, beta, C)
```

Rank-2k update of the Hermitian matrix C as alpha*A*B' + alpha'*B*A' + beta*C or alpha*A'*B + alpha'*B'*A + beta*C according to trans. The scalar beta has to be real. Only the uplo triangle of C is used. Return C.

LinearAlgebra.BLAS.her2k – Function.

```
her2k(uplo, trans, alpha, A, B)
```

Return the uplo triangle of alpha*A*B' + alpha'*B*A' or alpha*A'*B + alpha'*B'*A, according to trans.

```
her2k(uplo, trans, A, B)
```

Return the uplo triangle of A*B' + B*A' or A'*B + B'*A, according to trans.

LinearAlgebra.BLAS.trmm! – Function.

```
trmm!(side, ul, tA, dA, alpha, A, B)
```

Update B as alpha*A*B or one of the other three variants determined by side and tA. Only the ul triangle of A is used. dA determines if the diagonal values are read or are assumed to be all ones. Return the updated B.

LinearAlgebra.BLAS.trmm – Function.

```
trmm(side, ul, tA, dA, alpha, A, B)
```

Return alpha*A*B or one of the other three variants determined by side and tA. Only the ul triangle of A is used. dA determines if the diagonal values are read or are assumed to be all ones.

LinearAlgebra.BLAS.trsm! – Function.

```
trsm!(side, ul, tA, dA, alpha, A, B)
```

Overwrite B with the solution to A*X = alpha*B or one of the other three variants determined by side and tA. Only the ul triangle of A is used. dA determines if the diagonal values are read or are assumed to be all ones. Returns the updated B.

LinearAlgebra.BLAS.trsm – Function.

```
trsm(side, ul, tA, dA, alpha, A, B)
```

Return the solution to A*X = alpha*B or one of the other three variants determined by determined by side and tA. Only the ul triangle of A is used. dA determines if the diagonal values are read or are assumed to be all ones.

83.8 LAPACK functions

LinearAlgebra.LAPACK provides wrappers for some of the LAPACK functions for linear algebra. Those functions that overwrite one of the input arrays have names ending in '!'.

Usually a function has 4 methods defined, one each for Float64, Float32, ComplexF64 and ComplexF32 arrays.

Note that the LAPACK API provided by Julia can and will change in the future. Since this API is not user-facing, there is no commitment to support/deprecate this specific set of functions in future releases.

LinearAlgebra.LAPACK – Module.

Interfaces to LAPACK subroutines.

LinearAlgebra.LAPACK.gbtrf! – Function.

```
gbtrf!(kl, ku, m, AB) -> (AB, ipiv)
```

Compute the LU factorization of a banded matrix AB. kl is the first subdiagonal containing a nonzero band, ku is the last superdiagonal containing one, and m is the first dimension of the matrix AB. Returns the LU factorization in-place and ipiv, the vector of pivots used.

LinearAlgebra.LAPACK.gbtrs! – Function.

```
gbtrs!(trans, kl, ku, m, AB, ipiv, B)
```

Solve the equation AB * X = B. trans determines the orientation of AB. It may be N (no transpose), T (transpose), or C (conjugate transpose). kl is the first subdiagonal containing a nonzero band, ku is the last superdiagonal containing one, and m is the first dimension of the matrix AB. ipiv is the vector of pivots returned from gbtrf!. Returns the vector or matrix X, overwriting B in-place.

LinearAlgebra.LAPACK.gebal! – Function.

```
gebal!(job, A) -> (ilo, ihi, scale)
```

Balance the matrix `A` before computing its eigensystem or Schur factorization. `job` can be one of `N` (`A` will not be permuted or scaled), `P` (`A` will only be permuted), `S` (`A` will only be scaled), or `B` (`A` will be both permuted and scaled). Modifies `A` in-place and returns `ilo`, `ihi`, and `scale`. If permuting was turned on, `A[i,j] = 0` if `j > i` and `1 < j < ilo` or `j > ihi`. `scale` contains information about the scaling/permutations performed.

`LinearAlgebra.LAPACK.gebak!` – Function.

```
gebak!(job, side, ilo, ihi, scale, V)
```

Transform the eigenvectors `V` of a matrix balanced using `gebal!` to the unscaled/unpermuted eigenvectors of the original matrix. Modifies `V` in-place. `side` can be `L` (left eigenvectors are transformed) or `R` (right eigenvectors are transformed).

`LinearAlgebra.LAPACK.gebrd!` – Function.

```
gebrd!(A) -> (A, d, e, tauq, taup)
```

Reduce `A` in-place to bidiagonal form `A = QBP'`. Returns `A`, containing the bidiagonal matrix `B`; `d`, containing the diagonal elements of `B`; `e`, containing the off-diagonal elements of `B`; `tauq`, containing the elementary reflectors representing `Q`; and `taup`, containing the elementary reflectors representing `P`.

`LinearAlgebra.LAPACK.gelqf!` – Function.

```
gelqf!(A, tau)
```

Compute the `LQ` factorization of `A`, `A = LQ`. `tau` contains scalars which parameterize the elementary reflectors of the factorization. `tau` must have length greater than or equal to the smallest dimension of `A`.

Returns `A` and `tau` modified in-place.

```
gelqf!(A) -> (A, tau)
```

Compute the `LQ` factorization of `A`, `A = LQ`.

Returns `A`, modified in-place, and `tau`, which contains scalars which parameterize the elementary reflectors of the factorization.

`LinearAlgebra.LAPACK.geqlf!` – Function.

```
geqlf!(A, tau)
```

Compute the `QL` factorization of `A`, `A = QL`. `tau` contains scalars which parameterize the elementary reflectors of the factorization. `tau` must have length greater than or equal to the smallest dimension of `A`.

Returns `A` and `tau` modified in-place.

```
geqlf!(A) -> (A, tau)
```

Compute the QL factorization of A, A = QL.

Returns A, modified in-place, and tau, which contains scalars which parameterize the elementary reflectors of the factorization.

LinearAlgebra.LAPACK.geqrf! – Function.

```
geqrf!(A, tau)
```

Compute the QR factorization of A, A = QR. tau contains scalars which parameterize the elementary reflectors of the factorization. tau must have length greater than or equal to the smallest dimension of A.

Returns A and tau modified in-place.

```
geqrf!(A) -> (A, tau)
```

Compute the QR factorization of A, A = QR.

Returns A, modified in-place, and tau, which contains scalars which parameterize the elementary reflectors of the factorization.

LinearAlgebra.LAPACK.geqp3! – Function.

```
geqp3!(A, [jpvt, tau]) -> (A, tau, jpvt)
```

Compute the pivoted QR factorization of A, AP = QR using BLAS level 3. P is a pivoting matrix, represented by jpvt. tau stores the elementary reflectors. The arguments jpvt and tau are optional and allow for passing preallocated arrays. When passed, jpvt must have length greater than or equal to n if A is an (m x n) matrix and tau must have length greater than or equal to the smallest dimension of A. On entry, if jpvt[j] does not equal zero then the jth column of A is permuted to the front of AP.

A, jpvt, and tau are modified in-place.

LinearAlgebra.LAPACK.gerqf! – Function.

```
gerqf!(A, tau)
```

Compute the RQ factorization of A, A = RQ. tau contains scalars which parameterize the elementary reflectors of the factorization. tau must have length greater than or equal to the smallest dimension of A.

Returns A and tau modified in-place.

```
gerqf!(A) -> (A, tau)
```

Compute the RQ factorization of A, A = RQ.

Returns A, modified in-place, and tau, which contains scalars which parameterize the elementary reflectors of the factorization.

LinearAlgebra.LAPACK.geqrt! – Function.

```
geqrt!(A, T)
```

Compute the blocked QR factorization of A, A = QR. T contains upper triangular block reflectors which parameterize the elementary reflectors of the factorization. The first dimension of T sets the block size and it must be between 1 and n. The second dimension of T must equal the smallest dimension of A.

Returns A and T modified in-place.

```
geqrt!(A, nb) -> (A, T)
```

Compute the blocked QR factorization of A, A = QR. nb sets the block size and it must be between 1 and n, the second dimension of A.

Returns A, modified in-place, and T, which contains upper triangular block reflectors which parameterize the elementary reflectors of the factorization.

`LinearAlgebra.LAPACK.geqrt3!` – Function.

```
geqrt3!(A, T)
```

Recursively computes the blocked QR factorization of A, A = QR. T contains upper triangular block reflectors which parameterize the elementary reflectors of the factorization. The first dimension of T sets the block size and it must be between 1 and n. The second dimension of T must equal the smallest dimension of A.

Returns A and T modified in-place.

```
geqrt3!(A) -> (A, T)
```

Recursively computes the blocked QR factorization of A, A = QR.

Returns A, modified in-place, and T, which contains upper triangular block reflectors which parameterize the elementary reflectors of the factorization.

`LinearAlgebra.LAPACK.getrf!` – Function.

```
getrf!(A, ipiv) -> (A, ipiv, info)
```

Compute the pivoted LU factorization of A, A = LU. `ipiv` contains the pivoting information and `info` a code which indicates success (`info = 0`), a singular value in U (`info = i`, in which case `U[i,i]` is singular), or an error code (`info < 0`).

```
getrf!(A) -> (A, ipiv, info)
```

Compute the pivoted LU factorization of A, A = LU.

Returns A, modified in-place, `ipiv`, the pivoting information, and an `info` code which indicates success (`info = 0`), a singular value in U (`info = i`, in which case `U[i,i]` is singular), or an error code (`info < 0`).

`LinearAlgebra.LAPACK.tzrzf!` – Function.

```
tzrzf!(A) -> (A, tau)
```

Transforms the upper trapezoidal matrix A to upper triangular form in-place. Returns A and `tau`, the scalar parameters for the elementary reflectors of the transformation.

LinearAlgebra.LAPACK.ormrz! – Function.

```
ormrz!(side, trans, A, tau, C)
```

Multiplies the matrix C by Q from the transformation supplied by tzrzf!. Depending on side or trans the multiplication can be left-sided (side = L, Q*C) or right-sided (side = R, C*Q) and Q can be unmodified (trans = N), transposed (trans = T), or conjugate transposed (trans = C). Returns matrix C which is modified in-place with the result of the multiplication.

LinearAlgebra.LAPACK.gels! – Function.

```
gels!(trans, A, B) -> (F, B, ssr)
```

Solves the linear equation A * X = B, transpose(A) * X = B, or adjoint(A) * X = B using a QR or LQ factorization. Modifies the matrix/vector B in place with the solution. A is overwritten with its QR or LQ factorization. trans may be one of N (no modification), T (transpose), or C (conjugate transpose). gels! searches for the minimum norm/least squares solution. A may be under or over determined. The solution is returned in B.

LinearAlgebra.LAPACK.gesv! – Function.

```
gesv!(A, B) -> (B, A, ipiv)
```

Solves the linear equation A * X = B where A is a square matrix using the LU factorization of A. A is overwritten with its LU factorization and B is overwritten with the solution X. ipiv contains the pivoting information for the LU factorization of A.

LinearAlgebra.LAPACK.getrs! – Function.

```
getrs!(trans, A, ipiv, B)
```

Solves the linear equation A * X = B, transpose(A) * X = B, or adjoint(A) * X = B for square A. Modifies the matrix/vector B in place with the solution. A is the LU factorization from getrf!, with ipiv the pivoting information. trans may be one of N (no modification), T (transpose), or C (conjugate transpose).

LinearAlgebra.LAPACK.getri! – Function.

```
getri!(A, ipiv)
```

Computes the inverse of A, using its LU factorization found by getrf!. ipiv is the pivot information output and A contains the LU factorization of getrf!. A is overwritten with its inverse.

LinearAlgebra.LAPACK.gesvx! – Function.

```
gesvx!(fact, trans, A, AF, ipiv, equed, R, C, B) -> (X, equed, R, C, B, rcond, ferr, berr,
↪  work)
```

Solves the linear equation A * X = B (trans = N), transpose(A) * X = B (trans = T), or adjoint(A) * X = B (trans = C) using the LU factorization of A. fact may be E, in which case A will be equilibrated and copied to AF; F, in which case AF and ipiv from a previous LU factorization are inputs; or N, in which case A will be copied to AF and then factored. If fact = F, equed may be N, meaning A has not been equilibrated; R, meaning A was multiplied by Diagonal(R) from the left; C, meaning A was multiplied by Diagonal(C) from the right; or B, meaning A was multiplied by Diagonal(R) from the left and

Diagonal(C) from the right. If fact = F and equed = R or B the elements of R must all be positive. If fact = F and equed = C or B the elements of C must all be positive.

Returns the solution X; equed, which is an output if fact is not N, and describes the equilibration that was performed; R, the row equilibration diagonal; C, the column equilibration diagonal; B, which may be overwritten with its equilibrated form Diagonal(R)*B (if trans = N and equed = R,B) or Diagonal(C)*B (if trans = T,C and equed = C,B); rcond, the reciprocal condition number of A after equilbrating; ferr, the forward error bound for each solution vector in X; berr, the forward error bound for each solution vector in X; and work, the reciprocal pivot growth factor.

```
gesvx!(A, B)
```

The no-equilibration, no-transpose simplification of gesvx!.

LinearAlgebra.LAPACK.gelsd! – Function.

```
gelsd!(A, B, rcond) -> (B, rnk)
```

Computes the least norm solution of A * X = B by finding the SVD factorization of A, then dividing-and-conquering the problem. B is overwritten with the solution X. Singular values below rcond will be treated as zero. Returns the solution in B and the effective rank of A in rnk.

LinearAlgebra.LAPACK.gelsy! – Function.

```
gelsy!(A, B, rcond) -> (B, rnk)
```

Computes the least norm solution of A * X = B by finding the full QR factorization of A, then dividing-and-conquering the problem. B is overwritten with the solution X. Singular values below rcond will be treated as zero. Returns the solution in B and the effective rank of A in rnk.

LinearAlgebra.LAPACK.gglse! – Function.

```
gglse!(A, c, B, d) -> (X,res)
```

Solves the equation A * x = c where x is subject to the equality constraint B * x = d. Uses the formula ||c - A*x||^2 = 0 to solve. Returns X and the residual sum-of-squares.

LinearAlgebra.LAPACK.geev! – Function.

```
geev!(jobvl, jobvr, A) -> (W, VL, VR)
```

Finds the eigensystem of A. If jobvl = N, the left eigenvectors of A aren't computed. If jobvr = N, the right eigenvectors of A aren't computed. If jobvl = V or jobvr = V, the corresponding eigenvectors are computed. Returns the eigenvalues in W, the right eigenvectors in VR, and the left eigenvectors in VL.

LinearAlgebra.LAPACK.gesdd! – Function.

```
gesdd!(job, A) -> (U, S, VT)
```

Finds the singular value decomposition of A, A = U * S * V', using a divide and conquer approach. If job = A, all the columns of U and the rows of V' are computed. If job = N, no columns of U or rows of V' are computed. If job = O, A is overwritten with the columns of (thin) U and the rows of (thin) V'. If job = S, the columns of (thin) U and the rows of (thin) V' are computed and returned separately.

LinearAlgebra.LAPACK.gesvd! – Function.

```
gesvd!(jobu, jobvt, A) -> (U, S, VT)
```

Finds the singular value decomposition of A, A = U * S * V'. If jobu = A, all the columns of U are computed. If jobvt = A all the rows of V' are computed. If jobu = N, no columns of U are computed. If jobvt = N no rows of V' are computed. If jobu = O, A is overwritten with the columns of (thin) U. If jobvt = O, A is overwritten with the rows of (thin) V'. If jobu = S, the columns of (thin) U are computed and returned separately. If jobvt = S the rows of (thin) V' are computed and returned separately. jobu and jobvt can't both be O.

Returns U, S, and Vt, where S are the singular values of A.

LinearAlgebra.LAPACK.ggsvd! – Function.

```
ggsvd!(jobu, jobv, jobq, A, B) -> (U, V, Q, alpha, beta, k, l, R)
```

Finds the generalized singular value decomposition of A and B, U'*A*Q = D1*R and V'*B*Q = D2*R. D1 has alpha on its diagonal and D2 has beta on its diagonal. If jobu = U, the orthogonal/unitary matrix U is computed. If jobv = V the orthogonal/unitary matrix V is computed. If jobq = Q, the orthogonal/unitary matrix Q is computed. If jobu, jobv or jobq is N, that matrix is not computed. This function is only available in LAPACK versions prior to 3.6.0.

LinearAlgebra.LAPACK.ggsvd3! – Function.

```
ggsvd3!(jobu, jobv, jobq, A, B) -> (U, V, Q, alpha, beta, k, l, R)
```

Finds the generalized singular value decomposition of A and B, U'*A*Q = D1*R and V'*B*Q = D2*R. D1 has alpha on its diagonal and D2 has beta on its diagonal. If jobu = U, the orthogonal/unitary matrix U is computed. If jobv = V the orthogonal/unitary matrix V is computed. If jobq = Q, the orthogonal/unitary matrix Q is computed. If jobu, jobv, or jobq is N, that matrix is not computed. This function requires LAPACK 3.6.0.

LinearAlgebra.LAPACK.geevx! – Function.

```
geevx!(balanc, jobvl, jobvr, sense, A) -> (A, w, VL, VR, ilo, ihi, scale, abnrm, rconde,
↪  rcondv)
```

Finds the eigensystem of A with matrix balancing. If jobvl = N, the left eigenvectors of A aren't computed. If jobvr = N, the right eigenvectors of A aren't computed. If jobvl = V or jobvr = V, the corresponding eigenvectors are computed. If balanc = N, no balancing is performed. If balanc = P, A is permuted but not scaled. If balanc = S, A is scaled but not permuted. If balanc = B, A is permuted and scaled. If sense = N, no reciprocal condition numbers are computed. If sense = E, reciprocal condition numbers are computed for the eigenvalues only. If sense = V, reciprocal condition numbers are computed for the right eigenvectors only. If sense = B, reciprocal condition numbers are computed for the right eigenvectors and the eigenvectors. If sense = E,B, the right and left eigenvectors must be computed.

LinearAlgebra.LAPACK.ggev! – Function.

```
ggev!(jobvl, jobvr, A, B) -> (alpha, beta, vl, vr)
```

Finds the generalized eigendecomposition of A and B. If jobvl = N, the left eigenvectors aren't computed. If jobvr = N, the right eigenvectors aren't computed. If jobvl = V or jobvr = V, the corresponding eigenvectors are computed.

LinearAlgebra.LAPACK.ggev3! – Function.

```
ggev3!(jobvl, jobvr, A, B) -> (alpha, beta, vl, vr)
```

Finds the generalized eigendecomposition of A and B using a blocked algorithm. If jobvl = N, the left eigenvectors aren't computed. If jobvr = N, the right eigenvectors aren't computed. If jobvl = V or jobvr = V, the corresponding eigenvectors are computed. This function requires LAPACK 3.6.0.

LinearAlgebra.LAPACK.gtsv! – Function.

```
gtsv!(dl, d, du, B)
```

Solves the equation A * X = B where A is a tridiagonal matrix with dl on the subdiagonal, d on the diagonal, and du on the superdiagonal.

Overwrites B with the solution X and returns it.

LinearAlgebra.LAPACK.gttrf! – Function.

```
gttrf!(dl, d, du) -> (dl, d, du, du2, ipiv)
```

Finds the LU factorization of a tridiagonal matrix with dl on the subdiagonal, d on the diagonal, and du on the superdiagonal.

Modifies dl, d, and du in-place and returns them and the second superdiagonal du2 and the pivoting vector ipiv.

LinearAlgebra.LAPACK.gttrs! – Function.

```
gttrs!(trans, dl, d, du, du2, ipiv, B)
```

Solves the equation A * X = B (trans = N), transpose(A) * X = B (trans = T), or adjoint(A) * X = B (trans = C) using the LU factorization computed by gttrf!. B is overwritten with the solution X.

LinearAlgebra.LAPACK.orglq! – Function.

```
orglq!(A, tau, k = length(tau))
```

Explicitly finds the matrix Q of a LQ factorization after calling gelqf! on A. Uses the output of gelqf!. A is overwritten by Q.

LinearAlgebra.LAPACK.orgqr! – Function.

```
orgqr!(A, tau, k = length(tau))
```

Explicitly finds the matrix Q of a QR factorization after calling geqrf! on A. Uses the output of geqrf!. A is overwritten by Q.

LinearAlgebra.LAPACK.orgql! – Function.

```
orgql!(A, tau, k = length(tau))
```

Explicitly finds the matrix Q of a QL factorization after calling geqlf! on A. Uses the output of geqlf!. A is overwritten by Q.

LinearAlgebra.LAPACK.orgrq! – Function.

```
orgrq!(A, tau, k = length(tau))
```

Explicitly finds the matrix Q of a RQ factorization after calling gerqf! on A. Uses the output of gerqf!. A is overwritten by Q.

LinearAlgebra.LAPACK.ormlq! – Function.

```
ormlq!(side, trans, A, tau, C)
```

Computes Q * C (trans = N), transpose(Q) * C (trans = T), adjoint(Q) * C (trans = C) for side = L or the equivalent right-sided multiplication for side = R using Q from a LQ factorization of A computed using gelqf!. C is overwritten.

LinearAlgebra.LAPACK.ormqr! – Function.

```
ormqr!(side, trans, A, tau, C)
```

Computes Q * C (trans = N), transpose(Q) * C (trans = T), adjoint(Q) * C (trans = C) for side = L or the equivalent right-sided multiplication for side = R using Q from a QR factorization of A computed using geqrf!. C is overwritten.

LinearAlgebra.LAPACK.ormql! – Function.

```
ormql!(side, trans, A, tau, C)
```

Computes Q * C (trans = N), transpose(Q) * C (trans = T), adjoint(Q) * C (trans = C) for side = L or the equivalent right-sided multiplication for side = R using Q from a QL factorization of A computed using geqlf!. C is overwritten.

LinearAlgebra.LAPACK.ormrq! – Function.

```
ormrq!(side, trans, A, tau, C)
```

Computes Q * C (trans = N), transpose(Q) * C (trans = T), adjoint(Q) * C (trans = C) for side = L or the equivalent right-sided multiplication for side = R using Q from a RQ factorization of A computed using gerqf!. C is overwritten.

LinearAlgebra.LAPACK.gemqrt! – Function.

```
gemqrt!(side, trans, V, T, C)
```

Computes Q * C (trans = N), transpose(Q) * C (trans = T), adjoint(Q) * C (trans = C) for side = L or the equivalent right-sided multiplication for side = R using Q from a QR factorization of A computed using geqrt!. C is overwritten.

LinearAlgebra.LAPACK.posv! – Function.

```
posv!(uplo, A, B) -> (A, B)
```

Finds the solution to A * X = B where A is a symmetric or Hermitian positive definite matrix. If uplo = U the upper Cholesky decomposition of A is computed. If uplo = L the lower Cholesky decomposition of A is computed. A is overwritten by its Cholesky decomposition. B is overwritten with the solution X.

LinearAlgebra.LAPACK.potrf! – Function.

```
potrf!(uplo, A)
```

Computes the Cholesky (upper if uplo = U, lower if uplo = L) decomposition of positive-definite matrix A. A is overwritten and returned with an info code.

LinearAlgebra.LAPACK.potri! – Function.

```
potri!(uplo, A)
```

Computes the inverse of positive-definite matrix A after calling potrf! to find its (upper if uplo = U, lower if uplo = L) Cholesky decomposition.

A is overwritten by its inverse and returned.

LinearAlgebra.LAPACK.potrs! – Function.

```
potrs!(uplo, A, B)
```

Finds the solution to A * X = B where A is a symmetric or Hermitian positive definite matrix whose Cholesky decomposition was computed by potrf!. If uplo = U the upper Cholesky decomposition of A was computed. If uplo = L the lower Cholesky decomposition of A was computed. B is overwritten with the solution X.

LinearAlgebra.LAPACK.pstrf! – Function.

```
pstrf!(uplo, A, tol) -> (A, piv, rank, info)
```

Computes the (upper if uplo = U, lower if uplo = L) pivoted Cholesky decomposition of positive-definite matrix A with a user-set tolerance tol. A is overwritten by its Cholesky decomposition.

Returns A, the pivots piv, the rank of A, and an info code. If info = 0, the factorization succeeded. If info = i > 0, then A is indefinite or rank-deficient.

LinearAlgebra.LAPACK.ptsv! – Function.

```
ptsv!(D, E, B)
```

Solves A * X = B for positive-definite tridiagonal A. D is the diagonal of A and E is the off-diagonal. B is overwritten with the solution X and returned.

LinearAlgebra.LAPACK.pttrf! – Function.

```
pttrf!(D, E)
```

Computes the LDLt factorization of a positive-definite tridiagonal matrix with D as diagonal and E as off-diagonal. D and E are overwritten and returned.

LinearAlgebra.LAPACK.pttrs! – Function.

```
pttrs!(D, E, B)
```

Solves A * X = B for positive-definite tridiagonal A with diagonal D and off-diagonal E after computing A's LDLt factorization using pttrf!. B is overwritten with the solution X.

LinearAlgebra.LAPACK.trtri! – Function.

```
trtri!(uplo, diag, A)
```

Finds the inverse of (upper if uplo = U, lower if uplo = L) triangular matrix A. If diag = N, A has non-unit diagonal elements. If diag = U, all diagonal elements of A are one. A is overwritten with its inverse.

LinearAlgebra.LAPACK.trtrs! – Function.

```
trtrs!(uplo, trans, diag, A, B)
```

Solves A * X = B (trans = N), transpose(A) * X = B (trans = T), or adjoint(A) * X = B (trans = C) for (upper if uplo = U, lower if uplo = L) triangular matrix A. If diag = N, A has non-unit diagonal elements. If diag = U, all diagonal elements of A are one. B is overwritten with the solution X.

LinearAlgebra.LAPACK.trcon! – Function.

```
trcon!(norm, uplo, diag, A)
```

Finds the reciprocal condition number of (upper if uplo = U, lower if uplo = L) triangular matrix A. If diag = N, A has non-unit diagonal elements. If diag = U, all diagonal elements of A are one. If norm = I, the condition number is found in the infinity norm. If norm = 0 or 1, the condition number is found in the one norm.

LinearAlgebra.LAPACK.trevc! – Function.

```
trevc!(side, howmny, select, T, VL = similar(T), VR = similar(T))
```

Finds the eigensystem of an upper triangular matrix T. If side = R, the right eigenvectors are computed. If side = L, the left eigenvectors are computed. If side = B, both sets are computed. If howmny = A, all eigenvectors are found. If howmny = B, all eigenvectors are found and backtransformed using VL and VR. If howmny = S, only the eigenvectors corresponding to the values in select are computed.

LinearAlgebra.LAPACK.trrfs! – Function.

```
trrfs!(uplo, trans, diag, A, B, X, Ferr, Berr) -> (Ferr, Berr)
```

Estimates the error in the solution to A * X = B (trans = N), transpose(A) * X = B (trans = T), adjoint(A) * X = B (trans = C) for side = L, or the equivalent equations a right-handed side = R X * A after computing X using trtrs!. If uplo = U, A is upper triangular. If uplo = L, A is lower triangular. If diag = N, A has non-unit diagonal elements. If diag = U, all diagonal elements of A are one. Ferr and Berr are optional inputs. Ferr is the forward error and Berr is the backward error, each component-wise.

LinearAlgebra.LAPACK.stev! – Function.

```
stev!(job, dv, ev) -> (dv, Zmat)
```

Computes the eigensystem for a symmetric tridiagonal matrix with dv as diagonal and ev as off-diagonal. If job = N only the eigenvalues are found and returned in dv. If job = V then the eigenvectors are also found and returned in Zmat.

LinearAlgebra.LAPACK.stebz! – Function.

```
stebz!(range, order, vl, vu, il, iu, abstol, dv, ev) -> (dv, iblock, isplit)
```

Computes the eigenvalues for a symmetric tridiagonal matrix with `dv` as diagonal and `ev` as off-diagonal. If `range = A`, all the eigenvalues are found. If `range = V`, the eigenvalues in the half-open interval `(vl, vu]` are found. If `range = I`, the eigenvalues with indices between `il` and `iu` are found. If `order = B`, eigvalues are ordered within a block. If `order = E`, they are ordered across all the blocks. `abstol` can be set as a tolerance for convergence.

`LinearAlgebra.LAPACK.stegr!` – Function.

```
stegr!(jobz, range, dv, ev, vl, vu, il, iu) -> (w, Z)
```

Computes the eigenvalues (`jobz = N`) or eigenvalues and eigenvectors (`jobz = V`) for a symmetric tridiagonal matrix with `dv` as diagonal and `ev` as off-diagonal. If `range = A`, all the eigenvalues are found. If `range = V`, the eigenvalues in the half-open interval `(vl, vu]` are found. If `range = I`, the eigenvalues with indices between `il` and `iu` are found. The eigenvalues are returned in `w` and the eigenvectors in `Z`.

`LinearAlgebra.LAPACK.stein!` – Function.

```
stein!(dv, ev_in, w_in, iblock_in, isplit_in)
```

Computes the eigenvectors for a symmetric tridiagonal matrix with `dv` as diagonal and `ev_in` as off-diagonal. `w_in` specifies the input eigenvalues for which to find corresponding eigenvectors. `iblock_in` specifies the submatrices corresponding to the eigenvalues in `w_in`. `isplit_in` specifies the splitting points between the submatrix blocks.

`LinearAlgebra.LAPACK.syconv!` – Function.

```
syconv!(uplo, A, ipiv) -> (A, work)
```

Converts a symmetric matrix `A` (which has been factorized into a triangular matrix) into two matrices `L` and `D`. If `uplo = U`, `A` is upper triangular. If `uplo = L`, it is lower triangular. `ipiv` is the pivot vector from the triangular factorization. `A` is overwritten by `L` and `D`.

`LinearAlgebra.LAPACK.sysv!` – Function.

```
sysv!(uplo, A, B) -> (B, A, ipiv)
```

Finds the solution to `A * X = B` for symmetric matrix `A`. If `uplo = U`, the upper half of `A` is stored. If `uplo = L`, the lower half is stored. `B` is overwritten by the solution `X`. `A` is overwritten by its Bunch-Kaufman factorization. `ipiv` contains pivoting information about the factorization.

`LinearAlgebra.LAPACK.sytrf!` – Function.

```
sytrf!(uplo, A) -> (A, ipiv, info)
```

Computes the Bunch-Kaufman factorization of a symmetric matrix `A`. If `uplo = U`, the upper half of `A` is stored. If `uplo = L`, the lower half is stored.

Returns `A`, overwritten by the factorization, a pivot vector `ipiv`, and the error code `info` which is a non-negative integer. If `info` is positive the matrix is singular and the diagonal part of the factorization is exactly zero at position `info`.

```
sytrf!(uplo, A, ipiv) -> (A, ipiv, info)
```

Computes the Bunch-Kaufman factorization of a symmetric matrix `A`. If `uplo = U`, the upper half of `A` is stored. If `uplo = L`, the lower half is stored.

Returns `A`, overwritten by the factorization, the pivot vector `ipiv`, and the error code `info` which is a non-negative integer. If `info` is positive the matrix is singular and the diagonal part of the factorization is exactly zero at position `info`.

`LinearAlgebra.LAPACK.sytri!` – Function.

```
sytri!(uplo, A, ipiv)
```

Computes the inverse of a symmetric matrix `A` using the results of `sytrf!`. If `uplo = U`, the upper half of `A` is stored. If `uplo = L`, the lower half is stored. `A` is overwritten by its inverse.

`LinearAlgebra.LAPACK.sytrs!` – Function.

```
sytrs!(uplo, A, ipiv, B)
```

Solves the equation `A * X = B` for a symmetric matrix `A` using the results of `sytrf!`. If `uplo = U`, the upper half of `A` is stored. If `uplo = L`, the lower half is stored. `B` is overwritten by the solution `X`.

`LinearAlgebra.LAPACK.hesv!` – Function.

```
hesv!(uplo, A, B) -> (B, A, ipiv)
```

Finds the solution to `A * X = B` for Hermitian matrix `A`. If `uplo = U`, the upper half of `A` is stored. If `uplo = L`, the lower half is stored. `B` is overwritten by the solution `X`. `A` is overwritten by its Bunch-Kaufman factorization. `ipiv` contains pivoting information about the factorization.

`LinearAlgebra.LAPACK.hetrf!` – Function.

```
hetrf!(uplo, A) -> (A, ipiv, info)
```

Computes the Bunch-Kaufman factorization of a Hermitian matrix `A`. If `uplo = U`, the upper half of `A` is stored. If `uplo = L`, the lower half is stored.

Returns `A`, overwritten by the factorization, a pivot vector `ipiv`, and the error code `info` which is a non-negative integer. If `info` is positive the matrix is singular and the diagonal part of the factorization is exactly zero at position `info`.

```
hetrf!(uplo, A, ipiv) -> (A, ipiv, info)
```

Computes the Bunch-Kaufman factorization of a Hermitian matrix `A`. If `uplo = U`, the upper half of `A` is stored. If `uplo = L`, the lower half is stored.

Returns `A`, overwritten by the factorization, the pivot vector `ipiv`, and the error code `info` which is a non-negative integer. If `info` is positive the matrix is singular and the diagonal part of the factorization is exactly zero at position `info`.

`LinearAlgebra.LAPACK.hetri!` – Function.

```
hetri!(uplo, A, ipiv)
```

Computes the inverse of a Hermitian matrix `A` using the results of `sytrf!`. If `uplo = U`, the upper half of `A` is stored. If `uplo = L`, the lower half is stored. `A` is overwritten by its inverse.

`LinearAlgebra.LAPACK.hetrs!` – Function.

```
hetrs!(uplo, A, ipiv, B)
```

Solves the equation `A * X = B` for a Hermitian matrix `A` using the results of `sytrf!`. If `uplo = U`, the upper half of `A` is stored. If `uplo = L`, the lower half is stored. `B` is overwritten by the solution `X`.

`LinearAlgebra.LAPACK.syev!` – Function.

```
syev!(jobz, uplo, A)
```

Finds the eigenvalues (`jobz = N`) or eigenvalues and eigenvectors (`jobz = V`) of a symmetric matrix `A`. If `uplo = U`, the upper triangle of `A` is used. If `uplo = L`, the lower triangle of `A` is used.

`LinearAlgebra.LAPACK.syevr!` – Function.

```
syevr!(jobz, range, uplo, A, vl, vu, il, iu, abstol) -> (W, Z)
```

Finds the eigenvalues (`jobz = N`) or eigenvalues and eigenvectors (`jobz = V`) of a symmetric matrix `A`. If `uplo = U`, the upper triangle of `A` is used. If `uplo = L`, the lower triangle of `A` is used. If `range = A`, all the eigenvalues are found. If `range = V`, the eigenvalues in the half-open interval `(vl, vu]` are found. If `range = I`, the eigenvalues with indices between `il` and `iu` are found. `abstol` can be set as a tolerance for convergence.

The eigenvalues are returned in `W` and the eigenvectors in `Z`.

`LinearAlgebra.LAPACK.syevd!` – Function.

```
syevd!(jobz, uplo, A)
```

Finds the eigenvalues (`jobz = N`) or eigenvalues and eigenvectors (`jobz = V`) of a symmetric matrix `A`. If `uplo = U`, the upper triangle of `A` is used. If `uplo = L`, the lower triangle of `A` is used.

`LinearAlgebra.LAPACK.sygvd!` – Function.

```
sygvd!(itype, jobz, uplo, A, B) -> (w, A, B)
```

Finds the generalized eigenvalues (`jobz = N`) or eigenvalues and eigenvectors (`jobz = V`) of a symmetric matrix `A` and symmetric positive-definite matrix `B`. If `uplo = U`, the upper triangles of `A` and `B` are used. If `uplo = L`, the lower triangles of `A` and `B` are used. If `itype = 1`, the problem to solve is `A * x = lambda * B * x`. If `itype = 2`, the problem to solve is `A * B * x = lambda * x`. If `itype = 3`, the problem to solve is `B * A * x = lambda * x`.

`LinearAlgebra.LAPACK.bdsqr!` – Function.

```
bdsqr!(uplo, d, e_, Vt, U, C) -> (d, Vt, U, C)
```

Computes the singular value decomposition of a bidiagonal matrix with `d` on the diagonal and `e_` on the off-diagonal. If `uplo = U`, `e_` is the superdiagonal. If `uplo = L`, `e_` is the subdiagonal. Can optionally also compute the product `Q' * C`.

Returns the singular values in `d`, and the matrix `C` overwritten with `Q' * C`.

LinearAlgebra.LAPACK.bdsdc! – Function.

```
bdsdc!(uplo, compq, d, e_) -> (d, e, u, vt, q, iq)
```

Computes the singular value decomposition of a bidiagonal matrix with d on the diagonal and e_ on the off-diagonal using a divide and conqueq method. If uplo = U, e_ is the superdiagonal. If uplo = L, e_ is the subdiagonal. If compq = N, only the singular values are found. If compq = I, the singular values and vectors are found. If compq = P, the singular values and vectors are found in compact form. Only works for real types.

Returns the singular values in d, and if compq = P, the compact singular vectors in iq.

LinearAlgebra.LAPACK.gecon! – Function.

```
gecon!(normtype, A, anorm)
```

Finds the reciprocal condition number of matrix A. If normtype = I, the condition number is found in the infinity norm. If normtype = 0 or 1, the condition number is found in the one norm. A must be the result of getrf! and anorm is the norm of A in the relevant norm.

LinearAlgebra.LAPACK.gehrd! – Function.

```
gehrd!(ilo, ihi, A) -> (A, tau)
```

Converts a matrix A to Hessenberg form. If A is balanced with gebal! then ilo and ihi are the outputs of gebal!. Otherwise they should be ilo = 1 and ihi = size(A,2). tau contains the elementary reflectors of the factorization.

LinearAlgebra.LAPACK.orghr! – Function.

```
orghr!(ilo, ihi, A, tau)
```

Explicitly finds Q, the orthogonal/unitary matrix from gehrd!. ilo, ihi, A, and tau must correspond to the input/output to gehrd!.

LinearAlgebra.LAPACK.gees! – Function.

```
gees!(jobvs, A) -> (A, vs, w)
```

Computes the eigenvalues (jobvs = N) or the eigenvalues and Schur vectors (jobvs = V) of matrix A. A is overwritten by its Schur form.

Returns A, vs containing the Schur vectors, and w, containing the eigenvalues.

LinearAlgebra.LAPACK.gges! – Function.

```
gges!(jobvsl, jobvsr, A, B) -> (A, B, alpha, beta, vsl, vsr)
```

Computes the generalized eigenvalues, generalized Schur form, left Schur vectors (jobsvl = V), or right Schur vectors (jobvsr = V) of A and B.

The generalized eigenvalues are returned in alpha and beta. The left Schur vectors are returned in vsl and the right Schur vectors are returned in vsr.

LinearAlgebra.LAPACK.gges3! – Function.

```
gges3!(jobvsl, jobvsr, A, B) -> (A, B, alpha, beta, vsl, vsr)
```

Computes the generalized eigenvalues, generalized Schur form, left Schur vectors (`jobsvl = V`), or right Schur vectors (`jobvsr = V`) of `A` and `B` using a blocked algorithm. This function requires LAPACK 3.6.0.

The generalized eigenvalues are returned in `alpha` and `beta`. The left Schur vectors are returned in `vsl` and the right Schur vectors are returned in `vsr`.

`LinearAlgebra.LAPACK.trexc!` – Function.

```
trexc!(compq, ifst, ilst, T, Q) -> (T, Q)
trexc!(ifst, ilst, T, Q) -> (T, Q)
```

Reorder the Schur factorization `T` of a matrix, such that the diagonal block of `T` with row index `ifst` is moved to row index `ilst`. If `compq = V`, the Schur vectors `Q` are reordered. If `compq = N` they are not modified. The 4-arg method calls the 5-arg method with `compq = V`.

`LinearAlgebra.LAPACK.trsen!` – Function.

```
trsen!(job, compq, select, T, Q) -> (T, Q, w, s, sep)
trsen!(select, T, Q) -> (T, Q, w, s, sep)
```

Reorder the Schur factorization of a matrix and optionally finds reciprocal condition numbers. If `job = N`, no condition numbers are found. If `job = E`, only the condition number for this cluster of eigenvalues is found. If `job = V`, only the condition number for the invariant subspace is found. If `job = B` then the condition numbers for the cluster and subspace are found. If `compq = V` the Schur vectors `Q` are updated. If `compq = N` the Schur vectors are not modified. `select` determines which eigenvalues are in the cluster. The 3-arg method calls the 5-arg method with `job = N` and `compq = V`.

Returns `T`, `Q`, reordered eigenvalues in `w`, the condition number of the cluster of eigenvalues `s`, and the condition number of the invariant subspace `sep`.

`LinearAlgebra.LAPACK.tgsen!` – Function.

```
tgsen!(select, S, T, Q, Z) -> (S, T, alpha, beta, Q, Z)
```

Reorders the vectors of a generalized Schur decomposition. `select` specifies the eigenvalues in each cluster.

`LinearAlgebra.LAPACK.trsyl!` – Function.

```
trsyl!(transa, transb, A, B, C, isgn=1) -> (C, scale)
```

Solves the Sylvester matrix equation `A * X +/- X * B = scale*C` where `A` and `B` are both quasi-upper triangular. If `transa = N`, `A` is not modified. If `transa = T`, `A` is transposed. If `transa = C`, `A` is conjugate transposed. Similarly for `transb` and `B`. If `isgn = 1`, the equation `A * X + X * B = scale * C` is solved. If `isgn = -1`, the equation `A * X - X * B = scale * C` is solved.

Returns `X` (overwriting `C`) and `scale`.

`LinearAlgebra.LAPACK.hseqr!` – Function.

```
hseqr!(job, compz, ilo, ihi, H, Z) -> (H, Z, w)
```

Computes all eigenvalues and (optionally) the Schur factorization of a matrix reduced to Hessenberg form. If `H` is balanced with `gebal!` then `ilo` and `ihi` are the outputs of `gebal!`. Otherwise they should be `ilo = 1` and `ihi = size(H,2)`. `tau` contains the elementary reflectors of the factorization.

83.9 Developer Documentation

`LinearAlgebra.matprod_dest` – Function.

```
matprod_dest(A, B, T)
```

Return an appropriate `AbstractArray` with element type `T` that may be used to store the result of `A * B`.

Compat

This function requires at least Julia 1.11

`LinearAlgebra.haszero` – Function.

```
haszero(T::Type)
```

Return whether a type `T` has a unique zero element defined using `zero(T)`. If a type `M` specializes `zero(M)`, it may also choose to set `haszero(M)` to `true`. By default, `haszero` is assumed to be `false`, in which case the zero elements are deduced from values rather than the type.

Note

`haszero` is a conservative check that is used to dispatch to optimized paths. Extending it is optional, but encouraged.

Chapter 84

Logging

The Logging module provides a way to record the history and progress of a computation as a log of events. Events are created by inserting a logging statement into the source code, for example:

```
@warn "Abandon printf debugging, all ye who enter here!"
┌ Warning: Abandon printf debugging, all ye who enter here!
└ @ Main REPL[1]:1
```

The system provides several advantages over peppering your source code with calls to `println()`. First, it allows you to control the visibility and presentation of messages without editing the source code. For example, in contrast to the `@warn` above

```
@debug "The sum of some values $(sum(rand(100)))"
```

will produce no output by default. Furthermore, it's very cheap to leave debug statements like this in the source code because the system avoids evaluating the message if it would later be ignored. In this case `sum(rand(100))` and the associated string processing will never be executed unless debug logging is enabled.

Second, the logging tools allow you to attach arbitrary data to each event as a set of key–value pairs. This allows you to capture local variables and other program state for later analysis. For example, to attach the local array variable A and the sum of a vector v as the key s you can use

```
A = ones(Int, 4, 4)
v = ones(100)
@info "Some variables"  A  s=sum(v)

# output
┌ Info: Some variables
│   A =
│    4×4 Matrix{Int64}:
│     1  1  1  1
│     1  1  1  1
│     1  1  1  1
│     1  1  1  1
└   s = 100.0
```

All of the logging macros `@debug`, `@info`, `@warn` and `@error` share common features that are described in detail in the documentation for the more general macro `@logmsg`.

84.1 Log event structure

Each event generates several pieces of data, some provided by the user and some automatically extracted. Let's examine the user-defined data first:

- The *log level* is a broad category for the message that is used for early filtering. There are several standard levels of type `LogLevel`; user-defined levels are also possible. Each is distinct in purpose:
  - `Logging.Debug` (log level -1000) is information intended for the developer of the program. These events are disabled by default.
  - `Logging.Info` (log level 0) is for general information to the user. Think of it as an alternative to using `println` directly.
  - `Logging.Warn` (log level 1000) means something is wrong and action is likely required but that for now the program is still working.
  - `Logging.Error` (log level 2000) means something is wrong and it is unlikely to be recovered, at least by this part of the code. Often this log-level is unneeded as throwing an exception can convey all the required information.
- The *message* is an object describing the event. By convention `AbstractString`s passed as messages are assumed to be in markdown format. Other types will be displayed using `print(io, obj)` or `string(obj)` for text-based output and possibly `show(io,mime,obj)` for other multimedia displays used in the installed logger.
- Optional *key–value pairs* allow arbitrary data to be attached to each event. Some keys have conventional meaning that can affect the way an event is interpreted (see `@logmsg`).

The system also generates some standard information for each event:

- The `module` in which the logging macro was expanded.
- The `file` and `line` where the logging macro occurs in the source code.
- A message `id` that is a unique, fixed identifier for the *source code statement* where the logging macro appears. This identifier is designed to be fairly stable even if the source code of the file changes, as long as the logging statement itself remains the same.
- A `group` for the event, which is set to the base name of the file by default, without extension. This can be used to group messages into categories more finely than the log level (for example, all deprecation warnings have group `:depwarn`), or into logical groupings across or within modules.

Notice that some useful information such as the event time is not included by default. This is because such information can be expensive to extract and is also *dynamically* available to the current logger. It's simple to define a custom logger to augment event data with the time, backtrace, values of global variables and other useful information as required.

84.2 Processing log events

As you can see in the examples, logging statements make no mention of where log events go or how they are processed. This is a key design feature that makes the system composable and natural for concurrent use. It does this by separating two different concerns:

- *Creating* log events is the concern of the module author who needs to decide where events are triggered and which information to include.
- *Processing* of log events — that is, display, filtering, aggregation and recording — is the concern of the application author who needs to bring multiple modules together into a cooperating application.

Loggers

Processing of events is performed by a *logger*, which is the first piece of user configurable code to see the event. All loggers must be subtypes of `AbstractLogger`.

When an event is triggered, the appropriate logger is found by looking for a task-local logger with the global logger as fallback. The idea here is that the application code knows how log events should be processed and exists somewhere at the top of the call stack. So we should look up through the call stack to discover the logger — that is, the logger should be *dynamically scoped*. (This is a point of contrast with logging frameworks where the logger is *lexically scoped*; provided explicitly by the module author or as a simple global variable. In such a system it's awkward to control logging while composing functionality from multiple modules.)

The global logger may be set with `global_logger`, and task-local loggers controlled using `with_logger`. Newly spawned tasks inherit the logger of the parent task.

There are three logger types provided by the library. `ConsoleLogger` is the default logger you see when starting the REPL. It displays events in a readable text format and tries to give simple but user friendly control over formatting and filtering. `NullLogger` is a convenient way to drop all messages where necessary; it is the logging equivalent of the `devnull` stream. `SimpleLogger` is a very simplistic text formatting logger, mainly useful for debugging the logging system itself.

Custom loggers should come with overloads for the functions described in the reference section.

Early filtering and message handling

When an event occurs, a few steps of early filtering occur to avoid generating messages that will be discarded:

1. The message log level is checked against a global minimum level (set via `disable_logging`). This is a crude but extremely cheap global setting.
2. The current logger state is looked up and the message level checked against the logger's cached minimum level, as found by calling `Logging.min_enabled_level`. This behavior can be overridden via environment variables (more on this later).
3. The `Logging.shouldlog` function is called with the current logger, taking some minimal information (level, module, group, id) which can be computed statically. Most usefully, `shouldlog` is passed an event `id` which can be used to discard events early based on a cached predicate.

If all these checks pass, the message and key–value pairs are evaluated in full and passed to the current logger via the `Logging.handle_message` function. `handle_message()` may perform additional filtering as required and display the event to the screen, save it to a file, etc.

Exceptions that occur while generating the log event are captured and logged by default. This prevents individual broken events from crashing the application, which is helpful when enabling little-used debug events in a production system. This behavior can be customized per logger type by extending `Logging.catch_exceptions`.

84.3 Testing log events

Log events are a side effect of running normal code, but you might find yourself wanting to test particular informational messages and warnings. The `Test` module provides a `@test_logs` macro that can be used to pattern match against the log event stream.

84.4 Environment variables

Message filtering can be influenced through the `JULIA_DEBUG` environment variable, and serves as an easy way to enable debug logging for a file or module. Loading julia with `JULIA_DEBUG=loading` will activate `@debug` log messages in `loading.jl`. For example, in Linux shells:

```
$ JULIA_DEBUG=loading julia -e 'using OhMyREPL'
┌ Debug: Rejecting cache file /home/user/.julia/compiled/v0.7/OhMyREPL.ji due to it containing an
↪  incompatible cache header
└ @ Base loading.jl:1328
[ Info: Recompiling stale cache file /home/user/.julia/compiled/v0.7/OhMyREPL.ji for module
↪  OhMyREPL
┌ Debug: Rejecting cache file /home/user/.julia/compiled/v0.7/Tokenize.ji due to it containing an
↪  incompatible cache header
└ @ Base loading.jl:1328
...
```

On windows, the same can be achieved in `CMD` via first running `set JULIA_DEBUG="loading"` and in `Powershell` via `$env:JULIA_DEBUG="loading"`.

Similarly, the environment variable can be used to enable debug logging of modules, such as Pkg, or module roots (see `Base.moduleroot`). To enable all debug logging, use the special value `all`.

To turn debug logging on from the REPL, set `ENV["JULIA_DEBUG"]` to the name of the module of interest. Functions defined in the REPL belong to module `Main`; logging for them can be enabled like this:

```
julia> foo() = @debug "foo"
foo (generic function with 1 method)

julia> foo()

julia> ENV["JULIA_DEBUG"] = Main
Main

julia> foo()
┌ Debug: foo
```

```
└ @ Main REPL[1]:1
```

Use a comma separator to enable debug for multiple modules: `JULIA_DEBUG=loading,Main`.

84.5 Examples

Example: Writing log events to a file

Sometimes it can be useful to write log events to a file. Here is an example of how to use a task-local and global logger to write information to a text file:

```
# Load the logging module
julia> using Logging

# Open a textfile for writing
julia> io = open("log.txt", "w+")
IOStream(<file log.txt>)

# Create a simple logger
julia> logger = SimpleLogger(io)
SimpleLogger(IOStream(<file log.txt>), Info, Dict{Any,Int64}())

# Log a task-specific message
julia> with_logger(logger) do
           @info("a context specific log message")
       end

# Write all buffered messages to the file
julia> flush(io)

# Set the global logger to logger
julia> global_logger(logger)
SimpleLogger(IOStream(<file log.txt>), Info, Dict{Any,Int64}())

# This message will now also be written to the file
julia> @info("a global log message")

# Close the file
julia> close(io)
```

Example: Enable debug-level messages

Here is an example of creating a `ConsoleLogger` that lets through any messages with log level higher than, or equal, to `Logging.Debug`.

```
julia> using Logging

# Create a ConsoleLogger that prints any log messages with level >= Debug to stderr
julia> debuglogger = ConsoleLogger(stderr, Logging.Debug)
```

```
# Enable debuglogger for a task
julia> with_logger(debuglogger) do
           @debug "a context specific log message"
       end

# Set the global logger
julia> global_logger(debuglogger)
```

84.6 Reference

Logging module

`Logging.Logging` – Module.

Utilities for capturing, filtering and presenting streams of log events. Normally you don't need to import `Logging` to create log events; for this the standard logging macros such as `@info` are already exported by `Base` and available by default.

Creating events

`Base.CoreLogging.@logmsg` – Macro.

```
@debug message  [key=value | value ...]
@info  message  [key=value | value ...]
@warn  message  [key=value | value ...]
@error message  [key=value | value ...]

@logmsg level message [key=value | value ...]
```

Create a log record with an informational `message`. For convenience, four logging macros `@debug`, `@info`, `@warn` and `@error` are defined which log at the standard severity levels `Debug`, `Info`, `Warn` and `Error`. `@logmsg` allows `level` to be set programmatically to any `LogLevel` or custom log level types.

`message` should be an expression which evaluates to a string which is a human readable description of the log event. By convention, this string will be formatted as markdown when presented.

The optional list of `key=value` pairs supports arbitrary user defined metadata which will be passed through to the logging backend as part of the log record. If only a `value` expression is supplied, a key representing the expression will be generated using `Symbol`. For example, `x` becomes `x=x`, and `foo(10)` becomes `Symbol("foo(10)")=foo(10)`. For splatting a list of key value pairs, use the normal splatting syntax, `@info "blah" kws...`.

There are some keys which allow automatically generated log data to be overridden:

- `_module=mod` can be used to specify a different originating module from the source location of the message.
- `_group=symbol` can be used to override the message group (this is normally derived from the base name of the source file).
- `_id=symbol` can be used to override the automatically generated unique message identifier. This is useful if you need to very closely associate messages generated on different source lines.
- `_file=string` and `_line=integer` can be used to override the apparent source location of a log message.

There's also some key value pairs which have conventional meaning:

- `maxlog=integer` should be used as a hint to the backend that the message should be displayed no more than `maxlog` times.
- `exception=ex` should be used to transport an exception with a log message, often used with `@error`. An associated backtrace `bt` may be attached using the tuple `exception=(ex,bt)`.

Examples

```
@debug "Verbose debugging information.  Invisible by default"
@info  "An informational message"
@warn  "Something was odd.  You should pay attention"
@error "A non fatal error occurred"

x = 10
@info "Some variables attached to the message" x a=42.0

@debug begin
    sA = sum(A)
    "sum(A) = $sA is an expensive operation, evaluated only when `shouldlog` returns true"
end

for i=1:10000
    @info "With the default backend, you will only see (i = $i) ten times"  maxlog=10
    @debug "Algorithm1" i progress=i/10000
end
```

source

`Base.CoreLogging.LogLevel` – Type.

```
LogLevel(level)
```

Severity/verbosity of a log record.

The log level provides a key against which potential log records may be filtered, before any other work is done to construct the log record data structure itself.

Examples

```
julia> Logging.LogLevel(0) == Logging.Info
true
```

source

`Base.CoreLogging.Debug` – Constant.

```
Debug
```

Alias for `LogLevel(-1000)`.

source

`Base.CoreLogging.Info` – Constant.

```
Info
```

Alias for `LogLevel(0)`.

source

`Base.CoreLogging.Warn` – Constant.

```
Warn
```

Alias for `LogLevel(1000)`.

source

`Base.CoreLogging.Error` – Constant.

```
Error
```

Alias for `LogLevel(2000)`.

source

`Base.CoreLogging.BelowMinLevel` – Constant.

```
BelowMinLevel
```

Alias for `LogLevel(-1_000_001)`.

source

`Base.CoreLogging.AboveMaxLevel` – Constant.

```
AboveMaxLevel
```

Alias for `LogLevel(1_000_001)`.

source

Processing events with AbstractLogger

Event processing is controlled by overriding functions associated with `AbstractLogger`:

Methods to implement		Brief description
`Logging.handle_message`		Handle a log event
`Logging.shouldlog`		Early filtering of events
`Logging.min_enabled_level`		Lower bound for log level of accepted events
Optional methods	**Default definition**	**Brief description**
`Logging.catch_exceptions`	`true`	Catch exceptions during event evaluation

`Base.CoreLogging.AbstractLogger` – Type.

A logger controls how log records are filtered and dispatched. When a log record is generated, the logger is the first piece of user configurable code which gets to inspect the record and decide what to do with it.

source

Base.CoreLogging.handle_message – Function.

```
handle_message(logger, level, message, _module, group, id, file, line; key1=val1, ...)
```

Log a message to `logger` at `level`. The logical location at which the message was generated is given by module `_module` and `group`; the source location by `file` and `line`. `id` is an arbitrary unique value (typically a `Symbol`) to be used as a key to identify the log statement when filtering.

source

Base.CoreLogging.shouldlog – Function.

```
shouldlog(logger, level, _module, group, id)
```

Return `true` when `logger` accepts a message at `level`, generated for `_module`, `group` and with unique log identifier `id`.

source

Base.CoreLogging.min_enabled_level – Function.

```
min_enabled_level(logger)
```

Return the minimum enabled level for `logger` for early filtering. That is, the log level below or equal to which all messages are filtered.

source

Base.CoreLogging.catch_exceptions – Function.

```
catch_exceptions(logger)
```

Return `true` if the logger should catch exceptions which happen during log record construction. By default, messages are caught.

By default all exceptions are caught to prevent log message generation from crashing the program. This lets users confidently toggle little-used functionality - such as debug logging - in a production system.

If you want to use logging as an audit trail you should disable this for your logger type.

source

Base.CoreLogging.disable_logging – Function.

```
disable_logging(level)
```

Disable all log messages at log levels equal to or less than `level`. This is a *global* setting, intended to make debug logging extremely cheap when disabled. Note that this cannot be used to enable logging that is currently disabled by other mechanisms.

Examples

```
Logging.disable_logging(Logging.Info) # Disable debug and info
```

source

Using Loggers

Logger installation and inspection:

`Base.CoreLogging.global_logger` – Function.

```
global_logger()
```

Return the global logger, used to receive messages when no specific logger exists for the current task.

```
global_logger(logger)
```

Set the global logger to `logger`, and return the previous global logger.

source

`Base.CoreLogging.with_logger` – Function.

```
with_logger(function, logger)
```

Execute `function`, directing all log messages to `logger`.

Examples

```
function test(x)
    @info "x = $x"
end

with_logger(logger) do
    test(1)
    test([1,2])
end
```

source

`Base.CoreLogging.current_logger` – Function.

```
current_logger()
```

Return the logger for the current task, or the global logger if none is attached to the task.

source

Loggers that are supplied with the system:

`Base.CoreLogging.NullLogger` – Type.

```
NullLogger()
```

Logger which disables all messages and produces no output - the logger equivalent of /dev/null.

source

`Base.CoreLogging.ConsoleLogger` – Type.

```
ConsoleLogger([stream,] min_level=Info; meta_formatter=default_metafmt,
              show_limited=true, right_justify=0)
```

Logger with formatting optimized for readability in a text console, for example interactive work with the Julia REPL.

Log levels less than `min_level` are filtered out.

This Logger is thread-safe, with locks for both orchestration of message limits i.e. `maxlog`, and writes to the stream.

Message formatting can be controlled by setting keyword arguments:

- `meta_formatter` is a function which takes the log event metadata `(level, _module, group, id, file, line)` and returns a color (as would be passed to printstyled), prefix and suffix for the log message. The default is to prefix with the log level and a suffix containing the module, file and line location.
- `show_limited` limits the printing of large data structures to something which can fit on the screen by setting the `:limit IOContext` key during formatting.
- `right_justify` is the integer column which log metadata is right justified at. The default is zero (metadata goes on its own line).

source

`Base.CoreLogging.SimpleLogger` – Type.

```
SimpleLogger([stream,] min_level=Info)
```

Simplistic logger for logging all messages with level greater than or equal to `min_level` to `stream`. If stream is closed then messages with log level greater or equal to `Warn` will be logged to `stderr` and below to `stdout`.

This Logger is thread-safe, with a lock taken around orchestration of message limits i.e. `maxlog`, and writes to the stream.

source

Chapter 85

Markdown

This section describes Julia's markdown syntax, which is enabled by the Markdown standard library. The following Markdown elements are supported:

85.1 Inline elements

Here "inline" refers to elements that can be found within blocks of text, i.e. paragraphs. These include the following elements.

Bold

Surround words with two asterisks, **, to display the enclosed text in boldface.

```
A paragraph containing a **bold** word.
```

Italics

Surround words with one asterisk, *, to display the enclosed text in italics.

```
A paragraph containing an *italicized* word.
```

Literals

Surround text that should be displayed exactly as written with single backticks, ` .

```
A paragraph containing a `literal` word.
```

Literals should be used when writing text that refers to names of variables, functions, or other parts of a Julia program.

Tip

To include a backtick character within literal text use three backticks rather than one to enclose the text.

```
A paragraph containing ``` `backtick` characters ```.
```

By extension any odd number of backticks may be used to enclose a lesser number of backticks.

LaTeX

Surround text that should be displayed as mathematics using LaTeX syntax with double backticks, `` .

```
A paragraph containing some ``\LaTeX`` markup.
```

Tip

As with literals in the previous section, if literal backticks need to be written within double backticks use an even number greater than two. Note that if a single literal backtick needs to be included within LaTeX markup then two enclosing backticks is sufficient.

Note

The \ character should be escaped appropriately if the text is embedded in a Julia source code, for example, `"``\\LaTeX`` syntax in a docstring."`, since it is interpreted as a string literal. Alternatively, in order to avoid escaping, it is possible to use the `raw` string macro together with the `@doc` macro:

```
@doc raw"``\LaTeX`` syntax in a docstring." functionname
```

Links

Links to either external or internal targets can be written using the following syntax, where the text enclosed in square brackets, `[ ]`, is the name of the link and the text enclosed in parentheses, `( )`, is the URL.

```
A paragraph containing a link to [Julia](https://www.julialang.org).
```

It's also possible to add cross-references to other documented functions/methods/variables within the Julia documentation itself. For example:

```
"""
    tryparse(type, str; base)

Like [`parse`](@ref), but returns either a value of the requested type,
or [`nothing`](@ref) if the string does not contain a valid number.
"""
```

This will create a link in the generated docs to the `parse` documentation (which has more information about what this function actually does), and to the `nothing` documentation. It's good to include cross references to mutating/non-mutating versions of a function, or to highlight a difference between two similar-seeming functions.

Note

The above cross referencing is *not* a Markdown feature, and relies on Documenter.jl, which is used to build base Julia's documentation.

Footnote references

Named and numbered footnote references can be written using the following syntax. A footnote name must be a single alphanumeric word containing no punctuation.

```
A paragraph containing a numbered footnote [^1] and a named one [^named].
```

Note

The text associated with a footnote can be written anywhere within the same page as the footnote reference. The syntax used to define the footnote text is discussed in the Footnotes section below.

85.2 Toplevel elements

The following elements can be written either at the "toplevel" of a document or within another "toplevel" element.

Paragraphs

A paragraph is a block of plain text, possibly containing any number of inline elements defined in the Inline elements section above, with one or more blank lines above and below it.

```
This is a paragraph.

And this is *another* paragraph containing some emphasized text.
A new line, but still part of the same paragraph.
```

Headers

A document can be split up into different sections using headers. Headers use the following syntax:

```
# Level One
## Level Two
### Level Three
#### Level Four
##### Level Five
###### Level Six
```

A header line can contain any inline syntax in the same way as a paragraph can.

Tip

Try to avoid using too many levels of header within a single document. A heavily nested document may be indicative of a need to restructure it or split it into several pages covering separate topics.

Code blocks

Source code can be displayed as a literal block using an indent of four spaces or one tab as shown in the following example.

```
This is a paragraph.

    function func(x)
        # ...
    end

Another paragraph.
```

Additionally, code blocks can be enclosed using triple backticks with an optional "language" to specify how a block of code should be highlighted.

````
A code block without a "language":

```
function func(x)
    # ...
end
```

and another one with the "language" specified as `julia`:

```julia
function func(x)
    # ...
end
```
````

Note

"Fenced" code blocks, as shown in the last example, should be preferred over indented code blocks since there is no way to specify what language an indented code block is written in.

Block quotes

Text from external sources, such as quotations from books or websites, can be quoted using > characters prepended to each line of the quote as follows.

```
Here's a quote:

> Julia is a high-level, high-performance dynamic programming language for
> technical computing, with syntax that is familiar to users of other
> technical computing environments.
```

Note that a single space must appear after the > character on each line. Quoted blocks may themselves contain other toplevel or inline elements.

Images

The syntax for images is similar to the link syntax mentioned above. Prepending a `!` character to a link will display an image from the specified URL rather than a link to it.

```
![alternative text](link/to/image.png)
```

Lists

Unordered lists can be written by prepending each item in a list with either *, +, or -.

```
A list of items:

  * item one
  * item two
  * item three
```

Note the two spaces before each * and the single space after each one.

Lists can contain other nested toplevel elements such as lists, code blocks, or quoteblocks. A blank line should be left between each list item when including any toplevel elements within a list.

```
Another list:

  * item one

  * item two

    ```
    f(x) = x
    ```

  * And a sublist:

      + sub-item one
      + sub-item two
```

Note

The contents of each item in the list must line up with the first line of the item. In the above example the fenced code block must be indented by four spaces to align with the `i` in `item two`.

Ordered lists are written by replacing the "bullet" character, either *, +, or -, with a positive integer followed by either . or).

```
Two ordered lists:

 1. item one
 2. item two
 3. item three

 5) item five
 6) item six
 7) item seven
```

An ordered list may start from a number other than one, as in the second list of the above example, where it is numbered from five. As with unordered lists, ordered lists can contain nested toplevel elements.

Display equations

Large LaTeXequations that do not fit inline within a paragraph may be written as display equations using a fenced code block with the "language" `math` as in the example below.

````
```math
f(a) = \frac{1}{2\pi}\int_{0}^{2\pi} (\alpha+R\cos(\theta))d\theta
```
````

Footnotes

This syntax is paired with the inline syntax for Footnote references. Make sure to read that section as well.

Footnote text is defined using the following syntax, which is similar to footnote reference syntax, aside from the : character that is appended to the footnote label.

````
[^1]: Numbered footnote text.

[^note]:

    Named footnote text containing several toplevel elements
    indented by 4 spaces or one tab.

      * item one
      * item two
      * item three

    ```julia
    function func(x)
        # ...
    end
    ```
````

Note

No checks are done during parsing to make sure that all footnote references have matching footnotes.

Horizontal rules

The equivalent of an <hr> HTML tag can be achieved using three hyphens (`---`). For example:

```
Text above the line.

---

And text below the line.
```

Tables

Basic tables can be written using the syntax described below. Note that markdown tables have limited features and cannot contain nested toplevel elements unlike other elements discussed above – only inline elements are allowed. Tables must always contain a header row with column names. Cells cannot span multiple rows or columns of the table.

```
| Column One | Column Two | Column Three |
|:---------- | ---------- |:------------:|
| Row `1`    | Column `2` |              |
| *Row* 2    | **Row** 2  | Column ``3`` |
```

Note

As illustrated in the above example each column of `|` characters must be aligned vertically.

A `:` character on either end of a column's header separator (the row containing `-` characters) specifies whether the row is left-aligned, right-aligned, or (when `:` appears on both ends) center-aligned. Providing no `:` characters will default to right-aligning the column.

Admonitions

Specially formatted blocks, known as admonitions, can be used to highlight particular remarks. They can be defined using the following `!!!` syntax:

```
!!! note

    This is the content of the note.
    It is indented by 4 spaces. A tab would work as well.

!!! warning "Beware!"

    And this is another one.

    This warning admonition has a custom title: `"Beware!"`.
```

The first word after `!!!` declares the type of the admonition. There are standard admonition types that should produce special styling. Namely (in order of decreasing severity): `danger`, `warning`, `info`/`note`, and `tip`.

You can also use your own admonition types, as long as the type name only contains lowercase Latin characters (a-z). For example, you could have a `terminology` block like this:

```
!!! terminology "julia vs Julia"

    Strictly speaking, "Julia" refers to the language,
    and "julia" to the standard implementation.
```

However, unless the code rendering the Markdown special-cases that particular admonition type, it will get the default styling.

A custom title for the box can be provided as a string (in double quotes) after the admonition type. If no title text is specified after the admonition type, then the type name will be used as the title (e.g. `"Note"` for the `note` admonition).

Admonitions, like most other toplevel elements, can contain other toplevel elements (e.g. lists, images).

85.3 Markdown String Literals

The `md""` macro allows you to embed Markdown strings directly into your Julia code. This macro is designed to simplify the inclusion of Markdown-formatted text within your Julia source files.

Usage

```
result = md"This is a **custom** Markdown string with [a link](http://example.com)."
```

85.4 Markdown Syntax Extensions

Julia's markdown supports interpolation in a very similar way to basic string literals, with the difference that it will store the object itself in the Markdown tree (as opposed to converting it to a string). When the Markdown content is rendered the usual `show` methods will be called, and these can be overridden as usual. This design allows the Markdown to be extended with arbitrarily complex features (such as references) without cluttering the basic syntax.

In principle, the Markdown parser itself can also be arbitrarily extended by packages, or an entirely custom flavour of Markdown can be used, but this should generally be unnecessary.

85.5 API reference

`Markdown.MD` – Type.

```
MD
```

`MD` represents a Markdown document. Note that the `MD` constructor should not generally be used directly, since it constructs the internal data structures. Instead, you can construct `MD` objects using the exported macros `@md_str` and `@doc_str`.

`Markdown.@md_str` – Macro.

```
@md_str -> MD
```

Parse the given string as Markdown text and return a corresponding `MD` object.

See also `Markdown.parse`.

Examples

```
julia> s = md"# Hello, world!"
  Hello, world!
  ==============

julia> typeof(s)
Markdown.MD
```

`Markdown.@doc_str` – Macro.

```
@doc_str -> MD
```

Parse the given string as Markdown text, add line and module information and return a corresponding `MD` object.

`@doc_str` can be used in conjunction with the `Base.Docs` module. Please also refer to the manual section on documentation for more information.

Examples

```
julia> s = doc"f(x) = 2*x"
  f(x) = 2*x

julia> typeof(s)
Markdown.MD
```

`Markdown.parse` – Function.

```
parse(stream::IO)::MD
```

Parse the content of `stream` as Julia-flavored Markdown text and return the corresponding `MD` object.

```
Markdown.parse(markdown::AbstractString)::MD
```

Parse `markdown` as Julia-flavored Markdown text and return the corresponding `MD` object.

See also `@md_str`.

`Markdown.html` – Function.

```
html([io::IO], md)
```

Output the contents of the Markdown object `md` in HTML format, either writing to an (optional) `io` stream or returning a string.

One can alternatively use `show(io, "text/html", md)` or `repr("text/html", md)`, which differ in that they wrap the output in a `<div class="markdown"> ... </div>` element.

Examples

```
julia> html(md"hello _world_")
"<p>hello <em>world</em></p>\n"
```

`Markdown.latex` – Function.

```
latex([io::IO], md)
```

Output the contents of the Markdown object `md` in LaTeX format, either writing to an (optional) `io` stream or returning a string.

One can alternatively use `show(io, "text/latex", md)` or `repr("text/latex", md)`.

Examples

```
julia> latex(md"hello _world_")
"hello \\emph{world}\n\n"
```

Chapter 86

Memory-mapped I/O

Low level module for mmap (memory mapping of files).

`Mmap.Anonymous` – Type.

```
Mmap.Anonymous(name::AbstractString="", readonly::Bool=false, create::Bool=true)
```

Create an `IO`-like object for creating zeroed-out mmapped-memory that is not tied to a file for use in `mmap`. Used by `SharedArray` for creating shared memory arrays.

Examples

```
julia> using Mmap

julia> anon = Mmap.Anonymous();

julia> isreadable(anon)
true

julia> iswritable(anon)
true

julia> isopen(anon)
true
```

`Mmap.mmap` – Function.

```
mmap(io::Union{IOStream,AbstractString,Mmap.AnonymousMmap}[, type::Type{Array{T,N}}, dims,
↪  offset]; grow::Bool=true, shared::Bool=true)
mmap(type::Type{Array{T,N}}, dims)
```

Create an `Array` whose values are linked to a file, using memory-mapping. This provides a convenient way of working with data too large to fit in the computer's memory.

The type is an `Array{T,N}` with a bits-type element of `T` and dimension `N` that determines how the bytes of the array are interpreted. Note that the file must be stored in binary format, and no format conversions are possible (this is a limitation of operating systems, not Julia).

`dims` is a tuple or single `Integer` specifying the size or length of the array.

The file is passed via the stream argument, either as an open `IOStream` or filename string. When you initialize the stream, use `"r"` for a "read-only" array, and `"w+"` to create a new array used to write values to disk.

If no `type` argument is specified, the default is `Vector{UInt8}`.

Optionally, you can specify an offset (in bytes) if, for example, you want to skip over a header in the file. The default value for the offset is the current stream position for an `IOStream`.

The `grow` keyword argument specifies whether the disk file should be grown to accommodate the requested size of array (if the total file size is < requested array size). Write privileges are required to grow the file.

The `shared` keyword argument specifies whether the resulting `Array` and changes made to it will be visible to other processes mapping the same file.

For example, the following code

```
# Create a file for mmapping
# (you could alternatively use mmap to do this step, too)
using Mmap
A = rand(1:20, 5, 30)
s = open("/tmp/mmap.bin", "w+")
# We'll write the dimensions of the array as the first two Ints in the file
write(s, size(A,1))
write(s, size(A,2))
# Now write the data
write(s, A)
close(s)

# Test by reading it back in
s = open("/tmp/mmap.bin")   # default is read-only
m = read(s, Int)
n = read(s, Int)
A2 = mmap(s, Matrix{Int}, (m,n))
```

creates a `m`-by-`n` `Matrix{Int}`, linked to the file associated with stream `s`.

A more portable file would need to encode the word size – 32 bit or 64 bit – and endianness information in the header. In practice, consider encoding binary data using standard formats like HDF5 (which can be used with memory-mapping).

```
mmap(io, BitArray, [dims, offset])
```

Create a `BitArray` whose values are linked to a file, using memory-mapping; it has the same purpose, works in the same way, and has the same arguments, as `mmap`, but the byte representation is different.

Examples

```
julia> using Mmap

julia> io = open("mmap.bin", "w+");

julia> B = mmap(io, BitArray, (25,30000));
```

```
julia> B[3, 4000] = true;

julia> Mmap.sync!(B);

julia> close(io);

julia> io = open("mmap.bin", "r+");

julia> C = mmap(io, BitArray, (25,30000));

julia> C[3, 4000]
true

julia> C[2, 4000]
false

julia> close(io)

julia> rm("mmap.bin")
```

This creates a 25-by-30000 `BitArray`, linked to the file associated with stream `io`.

`Mmap.sync!` – Function.

```
Mmap.sync!(array)
```

Forces synchronization between the in-memory version of a memory-mapped `Array` or `BitArray` and the on-disk version.

Chapter 87

Network Options

`NetworkOptions.ca_roots` – Function.

```
ca_roots() :: Union{Nothing, String}
```

The `ca_roots()` function tells the caller where, if anywhere, to find a file or directory of PEM-encoded certificate authority roots. By default, on systems like Windows and macOS where the built-in TLS engines know how to verify hosts using the system's built-in certificate verification mechanism, this function will return `nothing`. On classic UNIX systems (excluding macOS), root certificates are typically stored in a file in `/etc`: the common places for the current UNIX system will be searched and if one of these paths exists, it will be returned; if none of these typical root certificate paths exist, then the path to the set of root certificates that are bundled with Julia is returned.

The default value returned by `ca_roots()` may be overridden by setting the `JULIA_SSL_CA_ROOTS_PATH`, `SSL_CERT_DIR`, or `SSL_CERT_FILE` environment variables, in which case this function will always return the value of the first of these variables that is set (whether the path exists or not). If `JULIA_SSL_CA_ROOTS_PATH` is set to the empty string, then the other variables are ignored (as if unset); if the other variables are set to the empty string, they behave is if they are not set.

`NetworkOptions.ca_roots_path` – Function.

```
ca_roots_path() :: String
```

The `ca_roots_path()` function is similar to the `ca_roots()` function except that it always returns a path to a file or directory of PEM-encoded certificate authority roots. When called on a system like Windows or macOS, where system root certificates are not stored in the file system, it will currently return the path to the set of root certificates that are bundled with Julia. (In the future, this function may instead extract the root certificates from the system and save them to a file whose path would be returned.)

If it is possible to configure a library that uses TLS to use the system certificates that is generally preferable: i.e. it is better to use `ca_roots()` which returns `nothing` to indicate that the system certs should be used. The `ca_roots_path()` function should only be used when configuring libraries which *require* a path to a file or directory for root certificates.

The default value returned by `ca_roots_path()` may be overridden by setting the `JULIA_SSL_CA_ROOTS_PATH`, `SSL_CERT_DIR`, or `SSL_CERT_FILE` environment variables, in which case this function will always return the value of the first of these variables that is set (whether the path exists or not). If `JULIA_SSL_CA_ROOTS_PATH`

is set to the empty string, then the other variables are ignored (as if unset); if the other variables are set to the empty string, they behave is if they are not set.

`NetworkOptions.ssh_dir` – Function.

```
ssh_dir() :: String
```

The `ssh_dir()` function returns the location of the directory where the `ssh` program keeps/looks for configuration files. By default this is `~/.ssh` but this can be overridden by setting the environment variable `SSH_DIR`.

`NetworkOptions.ssh_key_pass` – Function.

```
ssh_key_pass() :: String
```

The `ssh_key_pass()` function returns the value of the environment variable `SSH_KEY_PASS` if it is set or `nothing` if it is not set. In the future, this may be able to find a password by other means, such as secure system storage, so packages that need a password to decrypt an SSH private key should use this API instead of directly checking the environment variable so that they gain such capabilities automatically when they are added.

`NetworkOptions.ssh_key_name` – Function.

```
ssh_key_name() :: String
```

The `ssh_key_name()` function returns the base name of key files that SSH should use for when establishing a connection. There is usually no reason that this function should be called directly and libraries should generally use the `ssh_key_path` and `ssh_pub_key_path` functions to get full paths. If the environment variable `SSH_KEY_NAME` is set then this function returns that; otherwise it returns `id_rsa` by default.

`NetworkOptions.ssh_key_path` – Function.

```
ssh_key_path() :: String
```

The `ssh_key_path()` function returns the path of the SSH private key file that should be used for SSH connections. If the `SSH_KEY_PATH` environment variable is set then it will return that value. Otherwise it defaults to returning

```
joinpath(ssh_dir(), ssh_key_name())
```

This default value in turn depends on the `SSH_DIR` and `SSH_KEY_NAME` environment variables.

`NetworkOptions.ssh_pub_key_path` – Function.

```
ssh_pub_key_path() :: String
```

The `ssh_pub_key_path()` function returns the path of the SSH public key file that should be used for SSH connections. If the `SSH_PUB_KEY_PATH` environment variable is set then it will return that value. If that isn't set but `SSH_KEY_PATH` is set, it will return that path with the `.pub` suffix appended. If neither is set, it defaults to returning

```
joinpath(ssh_dir(), ssh_key_name() * ".pub")
```

This default value in turn depends on the SSH_DIR and SSH_KEY_NAME environment variables.

NetworkOptions.ssh_known_hosts_files – Function.

```
ssh_known_hosts_files() :: Vector{String}
```

The ssh_known_hosts_files() function returns a vector of paths of SSH known hosts files that should be used when establishing the identities of remote servers for SSH connections. By default this function returns

```
[joinpath(ssh_dir(), "known_hosts"), bundled_known_hosts]
```

where bundled_known_hosts is the path of a copy of a known hosts file that is bundled with this package (containing known hosts keys for github.com and gitlab.com). If the environment variable SSH_KNOWN_HOSTS_FILES is set, however, then its value is split into paths on the : character (or on ; on Windows) and this vector of paths is returned instead. If any component of this vector is empty, it is expanded to the default known hosts paths.

Packages that use ssh_known_hosts_files() should ideally look for matching entries by comparing the host name and key types, considering the first entry in any of the files which matches to be the definitive identity of the host. If the caller cannot compare the key type (e.g. because it has been hashes) then it must approximate the above algorithm by looking for all matching entries for a host in each file: if a file has any entries for a host then one of them must match; the caller should only continue to search further known hosts files if there are no entries for the host in question in an earlier file.

NetworkOptions.ssh_known_hosts_file – Function.

```
ssh_known_hosts_file() :: String
```

The ssh_known_hosts_file() function returns a single path of an SSH known hosts file that should be used when establishing the identities of remote servers for SSH connections. It returns the first path returned by ssh_known_hosts_files that actually exists. Callers who can look in more than one known hosts file should use ssh_known_hosts_files instead and look for host matches in all the files returned as described in that function's docs.

NetworkOptions.verify_host – Function.

```
verify_host(url::AbstractString, [transport::AbstractString]) :: Bool
```

The verify_host function tells the caller whether the identity of a host should be verified when communicating over secure transports like TLS or SSH. The url argument may be:

1. a proper URL staring with proto://
2. an ssh-style bare host name or host name prefixed with user@
3. an scp-style host as above, followed by : and a path location

In each case the host name part is parsed out and the decision about whether to verify or not is made based solely on the host name, not anything else about the input URL. In particular, the protocol of the URL does not matter (more below).

The `transport` argument indicates the kind of transport that the query is about. The currently known values are SSL/`ssl` (alias TLS/`tls`) and SSH/`ssh`. If the transport is omitted, the query will return `true` only if the host name should not be verified regardless of transport.

The host name is matched against the host patterns in the relevant environment variables depending on whether `transport` is supplied and what its value is:

- `JULIA_NO_VERIFY_HOSTS` — hosts that should not be verified for any transport
- `JULIA_SSL_NO_VERIFY_HOSTS` — hosts that should not be verified for SSL/TLS
- `JULIA_SSH_NO_VERIFY_HOSTS` — hosts that should not be verified for SSH
- `JULIA_ALWAYS_VERIFY_HOSTS` — hosts that should always be verified

The values of each of these variables is a comma-separated list of host name patterns with the following syntax — each pattern is split on `.` into parts and each part must one of:

1. A literal domain name component consisting of one or more ASCII letter, digit, hyphen or underscore (technically not part of a legal host name, but sometimes used). A literal domain name component matches only itself.
2. A **, which matches zero or more domain name components.
3. A *, which match any one domain name component.

When matching a host name against a pattern list in one of these variables, the host name is split on `.` into components and that sequence of words is matched against the pattern: a literal pattern matches exactly one host name component with that value; a * pattern matches exactly one host name component with any value; a ** pattern matches any number of host name components. For example:

- ** matches any host name
- **`.org` matches any host name in the `.org` top-level domain
- `example.com` matches only the exact host name `example.com`
- *`.example.com` matches `api.example.com` but not `example.com` or `v1.api.example.com`
- **`.example.com` matches any domain under `example.com`, including `example.com` itself, `api.example.com` and `v1.api.example.com`

Chapter 88

Pkg

Pkg is Julia's built-in package manager, and handles operations such as installing, updating and removing packages.

Note

What follows is a very brief introduction to Pkg. For more information on `Project.toml` files, `Manifest.toml` files, package version compatibility (`[compat]`), environments, registries, etc., it is highly recommended to read the full manual, which is available here: https://pkgdocs.julialang.org.

What follows is a quick overview of the basic features of Pkg. It should help new users become familiar with basic Pkg features such as adding and removing packages and working with environments.

Note

Some Pkg output is omitted in this section in order to keep this basic guide focused. This will help maintain a good pace and not get bogged down in details. If you require more details, refer to subsequent sections of the Pkg manual.

Note

This guide uses the Pkg REPL to execute Pkg commands. For non-interactive use, we recommend the Pkg API. The Pkg API is fully documented in the API Reference section of the Pkg documentation.

Pkg comes with a REPL. Enter the Pkg REPL by pressing `]` from the Julia REPL. To get back to the Julia REPL, press `Ctrl+C` or backspace (when the REPL cursor is at the beginning of the input).

Upon entering the Pkg REPL, you should see the following prompt:

```
(@v1.10) pkg>
```

To add a package, use add:

```
(@v1.10) pkg> add Example
   Resolving package versions...
   Installed Example — v0.5.3
    Updating `~/.julia/environments/v1.10/Project.toml`
```

```
  [7876af07] + Example v0.5.3
    Updating `~/.julia/environments/v1.10/Manifest.toml`
  [7876af07] + Example v0.5.3
```

After the package is installed, it can be loaded into the Julia session:

```
julia> import Example

julia> Example.hello("friend")
"Hello, friend"
```

We can also specify multiple packages at once to install:

```
(@v1.10) pkg> add JSON StaticArrays
```

The `status` command (or the shorter `st` command) can be used to see installed packages.

```
(@v1.10) pkg> st
Status `~/.julia/environments/v1.10/Project.toml`
  [7876af07] Example v0.5.3
  [682c06a0] JSON v0.21.3
  [90137ffa] StaticArrays v1.5.9
```

Note

Some Pkg REPL commands have a short and a long version of the command, for example `status` and `st`.

To remove packages, use `rm` (or `remove`):

```
(@v1.10) pkg> rm JSON StaticArrays
```

Use `up` (or `update`) to update the installed packages

```
(@v1.10) pkg> up
```

If you have been following this guide it is likely that the packages installed are at the latest version so `up` will not do anything. Below we show the status output in the case where we deliberately have installed an old version of the Example package and then upgrade it:

```
(@v1.10) pkg> st
Status `~/.julia/environments/v1.10/Project.toml`
^ [7876af07] Example v0.5.1
Info Packages marked with ^ have new versions available and may be upgradable.

(@v1.10) pkg> up
    Updating `~/.julia/environments/v1.10/Project.toml`
  [7876af07] ↑ Example v0.5.1 ⇒ v0.5.3
```

We can see that the status output tells us that there is a newer version available and that up upgrades the package.

For more information about managing packages, see the Managing Packages section of the documentation.

Up to this point, we have covered basic package management: adding, updating, and removing packages.

You may have noticed the `(@v1.10)` in the REPL prompt. This lets us know that `v1.10` is the **active environment**. Different environments can have totally different packages and versions installed from another environment. The active environment is the environment that will be modified by Pkg commands such as `add`, `rm` and `update`.

Let's set up a new environment so we may experiment. To set the active environment, use `activate`:

```
(@v1.10) pkg> activate tutorial
[ Info: activating new environment at `~/tutorial/Project.toml`.
```

Pkg lets us know we are creating a new environment and that this environment will be stored in the `~/tutorial` directory. The path to the environment is created relative to the current working directory of the REPL.

Pkg has also updated the REPL prompt in order to reflect the new active environment:

```
(tutorial) pkg>
```

We can ask for information about the active environment by using `status`:

```
(tutorial) pkg> status
    Status `~/tutorial/Project.toml`
   (empty environment)
```

`~/tutorial/Project.toml` is the location of the active environment's **project file**. A project file is a TOML file where Pkg stores the packages that have been explicitly installed. Notice this new environment is empty. Let us add some packages and observe:

```
(tutorial) pkg> add Example JSON
...

(tutorial) pkg> status
    Status `~/tutorial/Project.toml`
  [7876af07] Example v0.5.3
  [682c06a0] JSON v0.21.3
```

We can see that the `tutorial` environment now contains `Example` and `JSON`.

Note

If you have the same package (at the same version) installed in multiple environments, the package will only be downloaded and stored on the hard drive once. This makes environments very lightweight and effectively free to create. Using only the default environment with a huge number of packages in it is a common beginners mistake in Julia. Learning how to use environments effectively will improve your experience with Julia packages.

When you're done working in a specific environment and want to return to the default environment, use `activate` with no arguments:

```
(tutorial) pkg> activate
  Activating project at `~/.julia/environments/v1.10`

(@v1.10) pkg>
```

This returns you to the default `@v1.10` environment. There is no separate "deactivate" command—`activate` with no arguments serves this purpose.

For more information about environments, see the Working with Environments section of the documentation.

If you are ever stuck, you can ask Pkg for help:

```
(@v1.10) pkg> ?
```

You should see a list of available commands along with short descriptions. You can ask for more detailed help by specifying a command:

```
(@v1.10) pkg> ?develop
```

This guide should help you get started with Pkg. Pkg has much more to offer in terms of powerful package management. For more advanced topics, see Managing Packages, Working with Environments, and Creating Packages.

Chapter 89

Printf

The `Printf` module provides formatted output functions similar to the C standard library's `printf`. It allows formatted printing to an output stream or to a string.

`Printf.@printf` – Macro.

```
@printf([io::IO], "%Fmt", args...)
```

Print `args` using C `printf` style format specification string. Optionally, an `IO` may be passed as the first argument to redirect output.

Examples

```
julia> @printf "Hello %s" "world"
Hello world

julia> @printf "Scientific notation %e" 1.234
Scientific notation 1.234000e+00

julia> @printf "Scientific notation three digits %.3e" 1.23456
Scientific notation three digits 1.235e+00

julia> @printf "Decimal two digits %.2f" 1.23456
Decimal two digits 1.23

julia> @printf "Padded to length 5 %5i" 123
Padded to length 5   123

julia> @printf "Padded with zeros to length 6 %06i" 123
Padded with zeros to length 6 000123

julia> @printf "Use shorter of decimal or scientific %g %g" 1.23 12300000.0
Use shorter of decimal or scientific 1.23 1.23e+07

julia> @printf "Use dynamic width and precision  %*.*f" 10 2 0.12345
Use dynamic width and precision        0.12
```

For a systematic specification of the format, see here. See also `@sprintf` to get the result as a `String` instead of it being printed.

Caveats

`Inf` and `NaN` are printed consistently as `Inf` and `NaN` for flags `%a`, `%A`, `%e`, `%E`, `%f`, `%F`, `%g`, and `%G`. Furthermore, if a floating point number is equally close to the numeric values of two possible output strings, the output string further away from zero is chosen.

Examples

```
julia> @printf("%f %F %f %F", Inf, Inf, NaN, NaN)
Inf Inf NaN NaN

julia> @printf "%.0f %.1f %f" 0.5 0.025 -0.0078125
0 0.0 -0.007812
```

Julia 1.8

Starting in Julia 1.8, `%s` (string) and `%c` (character) widths are computed using `textwidth`, which e.g. ignores zero-width characters (such as combining characters for diacritical marks) and treats certain "wide" characters (e.g. emoji) as width 2.

Julia 1.10

Dynamic width specifiers like `%*s` and `%0*.*f` require Julia 1.10.

`Printf.@sprintf` – Macro.

```
@sprintf("%Fmt", args...)
```

Return `@printf` formatted output as string.

Examples

```
julia> @sprintf "this is a %s %15.1f" "test" 34.567
"this is a test            34.6"
```

`Printf.Format` – Type.

```
Printf.Format(format_str)
```

Create a C printf-compatible format object that can be used for formatting values.

The input `format_str` can include any valid format specifier character and modifiers.

A `Format` object can be passed to `Printf.format(f::Format, args...)` to produce a formatted string, or `Printf.format(io::IO, f::Format, args...)` to print the formatted string directly to `io`.

For convenience, the `Printf.format"..."` string macro form can be used for building a `Printf.Format` object at macro-expansion-time.

Julia 1.6

`Printf.Format` requires Julia 1.6 or later.

`Printf.format` – Function.

```
Printf.format(f::Printf.Format, args...) => String
Printf.format(io::IO, f::Printf.Format, args...)
```

Apply a printf format object `f` to provided `args` and return the formatted string (1st method), or print directly to an `io` object (2nd method). See `@printf` for more details on C `printf` support.

Chapter 90

Profiling

90.1 CPU Profiling

There are two main approaches to CPU profiling julia code:

90.2 Via `@profile`

Where profiling is enabled for a given call via the `@profile` macro.

```
julia> using Profile

julia> @profile foo()

julia> Profile.print()
Overhead ╎ [+additional indent] Count File:Line; Function
=========================================================
    ╎147  @Base/client.jl:506; _start()
        ╎ 147  @Base/client.jl:318; exec_options(opts::Base.JLOptions)
...
```

90.3 Triggered During Execution

Tasks that are already running can also be profiled for a fixed time period at any user-triggered time.

To trigger the profiling:

- MacOS & FreeBSD (BSD-based platforms): Use `ctrl-t` or pass a `SIGINFO` signal to the julia process i.e. `% kill -INFO $julia_pid`
- Linux: Pass a `SIGUSR1` signal to the julia process i.e. `% kill -USR1 $julia_pid`
- Windows: Not currently supported.

First, a single stack trace at the instant that the signal was thrown is shown, then a 1 second profile is collected, followed by the profile report at the next yield point, which may be at task completion for code without yield points e.g. tight loops.

Optionally set environment variable JULIA_PROFILE_PEEK_HEAP_SNAPSHOT to 1 to also automatically collect a heap snapshot.

```
julia> foo()
##== the user sends a trigger while foo is running ==##
load: 2.53  cmd: julia 88903 running 6.16u 0.97s

================================================================================
Information request received. A stacktrace will print followed by a 1.0 second profile
================================================================================

signal (29): Information request: 29
__psynch_cvwait at /usr/lib/system/libsystem_kernel.dylib (unknown line)
_pthread_cond_wait at /usr/lib/system/libsystem_pthread.dylib (unknown line)
...

=====================================================================
Profile collected. A report will print if the Profile module is loaded
=====================================================================

Overhead ╎ [+additional indent] Count File:Line; Function
=========================================================
Thread 1 Task 0x000000011687c010 Total snapshots: 572. Utilization: 100%
   ╎147 @Base/client.jl:506; _start()
       ╎ 147 @Base/client.jl:318; exec_options(opts::Base.JLOptions)
...

Thread 2 Task 0x0000000116960010 Total snapshots: 572. Utilization: 0%
   ╎572 @Base/task.jl:587; task_done_hook(t::Task)
      ╎ 572 @Base/task.jl:879; wait()
...
```

Customization

The duration of the profiling can be adjusted via Profile.set_peek_duration

The profile report is broken down by thread and task. Pass a no-arg function to Profile.peek_report[] to override this. i.e. Profile.peek_report[] = () -> Profile.print() to remove any grouping. This could also be overridden by an external profile data consumer.

90.4 Reference

Profile.@profile – Macro.

```
@profile
```

@profile <expression> runs your expression while taking periodic backtraces. These are appended to an internal buffer of backtraces.

The methods in Profile are not exported and need to be called e.g. as Profile.print().

Profile.clear – Function.

```
clear()
```

Clear any existing backtraces from the internal buffer.

`Profile.print` – Function.

```
print([io::IO = stdout,] [data::Vector = fetch()], [lidict::Union{LineInfoDict,
↪  LineInfoFlatDict} = getdict(data)]; kwargs...)
print(path::String, [cols::Int = 1000], [data::Vector = fetch()], [lidict::Union{LineInfoDict,
↪  LineInfoFlatDict} = getdict(data)]; kwargs...)
```

Prints profiling results to `io` (by default, `stdout`). If you do not supply a `data` vector, the internal buffer of accumulated backtraces will be used. Paths are clickable links in supported terminals and specialized for `JULIA_EDITOR` with line numbers, or just file links if no editor is set.

The keyword arguments can be any combination of:

- `format` – Determines whether backtraces are printed with (default, `:tree`) or without (`:flat`) indentation indicating tree structure.
- `C` – If `true`, backtraces from C and Fortran code are shown (normally they are excluded).
- `combine` – If `true` (default), instruction pointers are merged that correspond to the same line of code.
- `maxdepth` – Limits the depth higher than `maxdepth` in the `:tree` format.
- `sortedby` – Controls the order in `:flat` format. `:filefuncline` (default) sorts by the source line, `:count` sorts in order of number of collected samples, and `:overhead` sorts by the number of samples incurred by each function by itself.
- `groupby` – Controls grouping over tasks and threads, or no grouping. Options are `:none` (default), `:thread`, `:task`, `[:thread, :task]`, or `[:task, :thread]` where the last two provide nested grouping.
- `noisefloor` – Limits frames that exceed the heuristic noise floor of the sample (only applies to format `:tree`). A suggested value to try for this is 2.0 (the default is 0). This parameter hides samples for which `n <= noisefloor * √N`, where `n` is the number of samples on this line, and `N` is the number of samples for the callee.
- `mincount` – Limits the printout to only those lines with at least `mincount` occurrences.
- `recur` – Controls the recursion handling in `:tree` format. `:off` (default) prints the tree as normal. `:flat` instead compresses any recursion (by ip), showing the approximate effect of converting any self-recursion into an iterator. `:flatc` does the same but also includes collapsing of C frames (may do odd things around `jl_apply`).
- `threads::Union{Int,AbstractVector{Int}}` – Specify which threads to include snapshots from in the report. Note that this does not control which threads samples are collected on (which may also have been collected on another machine).
- `tasks::Union{Int,AbstractVector{Int}}` – Specify which tasks to include snapshots from in the report. Note that this does not control which tasks samples are collected within.

Julia 1.8

The `groupby`, `threads`, and `tasks` keyword arguments were introduced in Julia 1.8.

> **Note**
>
> Profiling on windows is limited to the main thread. Other threads have not been sampled and will not show in the report.

```
print([io::IO = stdout,] data::Vector, lidict::LineInfoDict; kwargs...)
```

Prints profiling results to `io`. This variant is used to examine results exported by a previous call to `retrieve`. Supply the vector `data` of backtraces and a dictionary `lidict` of line information.

See `Profile.print([io], data)` for an explanation of the valid keyword arguments.

`Profile.init` – Function.

```
init(; n::Integer, delay::Real)
```

Configure the `delay` between backtraces (measured in seconds), and the number `n` of instruction pointers that may be stored per thread. Each instruction pointer corresponds to a single line of code; backtraces generally consist of a long list of instruction pointers. Note that 6 spaces for instruction pointers per backtrace are used to store metadata and two NULL end markers. Current settings can be obtained by calling this function with no arguments, and each can be set independently using keywords or in the order `(n, delay)`.

`Profile.fetch` – Function.

```
fetch(;include_meta = true) -> data
```

Return a copy of the buffer of profile backtraces. Note that the values in `data` have meaning only on this machine in the current session, because it depends on the exact memory addresses used in JIT-compiling. This function is primarily for internal use; `retrieve` may be a better choice for most users. By default metadata such as threadid and taskid is included. Set `include_meta` to `false` to strip metadata.

`Profile.retrieve` – Function.

```
retrieve(; kwargs...) -> data, lidict
```

"Exports" profiling results in a portable format, returning the set of all backtraces (`data`) and a dictionary that maps the (session-specific) instruction pointers in `data` to `LineInfo` values that store the file name, function name, and line number. This function allows you to save profiling results for future analysis.

`Profile.callers` – Function.

```
callers(funcname, [data, lidict], [filename=<filename>],
↪  [linerange=<start:stop>])::Vector{Tuple{count, lineinfo}}
```

Given a previous profiling run, determine who called a particular function. Supplying the filename (and optionally, range of line numbers over which the function is defined) allows you to disambiguate an overloaded method. The returned value is a vector containing a count of the number of calls and line information about the caller. One can optionally supply backtrace `data` obtained from `retrieve`; otherwise, the current internal profile buffer is used.

`Profile.clear_malloc_data` – Function.

```
clear_malloc_data()
```

Clears any stored memory allocation data when running julia with `--track-allocation`. Execute the command(s) you want to test (to force JIT-compilation), then call `clear_malloc_data`. Then execute your command(s) again, quit Julia, and examine the resulting `*.mem` files.

`Profile.get_peek_duration` – Function.

```
get_peek_duration()
```

Get the duration in seconds of the profile "peek" that is triggered via `SIGINFO` or `SIGUSR1`, depending on platform.

`Profile.set_peek_duration` – Function.

```
set_peek_duration(t::Float64)
```

Set the duration in seconds of the profile "peek" that is triggered via `SIGINFO` or `SIGUSR1`, depending on platform.

90.5 Memory profiling

`Profile.Allocs.@profile` – Macro.

```
Profile.Allocs.@profile [sample_rate=0.1] expr
```

Profile allocations that happen during `expr`, returning both the result and AllocResults struct.

A sample rate of 1.0 will record everything; 0.0 will record nothing.

```
julia> Profile.Allocs.@profile sample_rate=0.01 peakflops()
1.03733270279065e11

julia> results = Profile.Allocs.fetch()

julia> last(sort(results.allocs, by=x->x.size))
Profile.Allocs.Alloc(Vector{Any}, Base.StackTraces.StackFrame[_new_array_ at array.c:127,
↪  ...], 5576)
```

See the profiling tutorial in the Julia documentation for more information.

Julia 1.11

Older versions of Julia could not capture types in all cases. In older versions of Julia, if you see an allocation of type `Profile.Allocs.UnknownType`, it means that the profiler doesn't know what type of object was allocated. This mainly happened when the allocation was coming from generated code produced by the compiler. See issue #43688 for more info.

Since Julia 1.11, all allocations should have a type reported.

Julia 1.8

The allocation profiler was added in Julia 1.8.

The methods in `Profile.Allocs` are not exported and need to be called e.g. as `Profile.Allocs.fetch()`.

`Profile.Allocs.clear` – Function.

```
Profile.Allocs.clear()
```

Clear all previously profiled allocation information from memory.

`Profile.Allocs.print` – Function.

```
Profile.Allocs.print([io::IO = stdout,] [data::AllocResults = fetch()]; kwargs...)
```

Prints profiling results to `io` (by default, `stdout`). If you do not supply a `data` vector, the internal buffer of accumulated backtraces will be used.

See `Profile.print` for an explanation of the valid keyword arguments.

`Profile.Allocs.fetch` – Function.

```
Profile.Allocs.fetch()
```

Retrieve the recorded allocations, and decode them into Julia objects which can be analyzed.

`Profile.Allocs.start` – Function.

```
Profile.Allocs.start(sample_rate::Real)
```

Begin recording allocations with the given sample rate A sample rate of 1.0 will record everything; 0.0 will record nothing.

`Profile.Allocs.stop` – Function.

```
Profile.Allocs.stop()
```

Stop recording allocations.

90.6 Heap Snapshots

`Profile.take_heap_snapshot` – Function.

```
Profile.take_heap_snapshot(filepath::String, all_one::Bool=false;
                           redact_data::Bool=true, streaming::Bool=false)
Profile.take_heap_snapshot(all_one::Bool=false; redact_data:Bool=true,
                           dir::String=nothing, streaming::Bool=false)
```

Write a snapshot of the heap, in the JSON format expected by the Chrome Devtools Heap Snapshot viewer (.heapsnapshot extension) to a file (`$pid_$timestamp.heapsnapshot`) in the current directory by default (or tempdir if the current directory is unwritable), or in `dir` if given, or the given full file path, or IO stream.

If `all_one` is true, then report the size of every object as one so they can be easily counted. Otherwise, report the actual size.

If `redact_data` is true (default), then do not emit the contents of any object.

If `streaming` is true, we will stream the snapshot data out into four files, using filepath as the prefix, to avoid having to hold the entire snapshot in memory. This option should be used for any setting where your memory is constrained. These files can then be reassembled by calling Profile.HeapSnapshot.assemble_snapshot(), which can be done offline.

NOTE: We strongly recommend setting streaming=true for performance reasons. Reconstructing the snapshot from the parts requires holding the entire snapshot in memory, so if the snapshot is large, you can run out of memory while processing it. Streaming allows you to reconstruct the snapshot offline, after your workload is done running. If you do attempt to collect a snapshot with streaming=false (the default, for backwards-compatibility) and your process is killed, note that this will always save the parts in the same directory as your provided filepath, so you can still reconstruct the snapshot after the fact, via `assemble_snapshot()`.

The methods in `Profile` are not exported and need to be called e.g. as `Profile.take_heap_snapshot()`.

```
julia> using Profile

julia> Profile.take_heap_snapshot("snapshot.heapsnapshot")
```

Traces and records julia objects on the heap. This only records objects known to the Julia garbage collector. Memory allocated by external libraries not managed by the garbage collector will not show up in the snapshot.

To avoid OOMing while recording the snapshot, we added a streaming option to stream out the heap snapshot into four files,

```
julia> using Profile

julia> Profile.take_heap_snapshot("snapshot"; streaming=true)
```

where "snapshot" is the filepath as the prefix for the generated files.

Once the snapshot files are generated, they could be assembled offline with the following command:

```
julia> using Profile

julia> Profile.HeapSnapshot.assemble_snapshot("snapshot", "snapshot.heapsnapshot")
```

The resulting heap snapshot file can be uploaded to chrome devtools to be viewed. For more information, see the chrome devtools docs. An alternative for analyzing Chromium heap snapshots is with the VS Code extension `ms-vscode.vscode-js-profile-flame`.

The Firefox heap snapshots are of a different format, and Firefox currently may *not* be used for viewing the heap snapshots generated by Julia.

Chapter 91

Random Numbers

Random number generation in Julia uses the Xoshiro256++ algorithm by default, with per-`Task` state. Other RNG types can be plugged in by inheriting the `AbstractRNG` type; they can then be used to obtain multiple streams of random numbers.

The PRNGs (pseudorandom number generators) exported by the `Random` package are:

- `TaskLocalRNG`: a token that represents use of the currently active Task-local stream, deterministically seeded from the parent task, or by `RandomDevice` (with system randomness) at program start
- `Xoshiro`: generates a high-quality stream of random numbers with a small state vector and high performance using the Xoshiro256++ algorithm
- `RandomDevice`: for OS-provided entropy. This may be used for cryptographically secure random numbers (CS(P)RNG).
- `MersenneTwister`: an alternate high-quality PRNG which was the default in older versions of Julia, and is also quite fast, but requires much more space to store the state vector and generate a random sequence.

Most functions related to random generation accept an optional `AbstractRNG` object as first argument. Some also accept dimension specifications `dims...` (which can also be given as a tuple) to generate arrays of random values. In a multi-threaded program, you should generally use different RNG objects from different threads or tasks in order to be thread-safe. However, the default RNG is thread-safe as of Julia 1.3 (using a per-thread RNG up to version 1.6, and per-task thereafter).

The provided RNGs can generate uniform random numbers of the following types: `Float16`, `Float32`, `Float64`, `BigFloat`, `Bool`, `Int8`, `UInt8`, `Int16`, `UInt16`, `Int32`, `UInt32`, `Int64`, `UInt64`, `Int128`, `UInt128`, `BigInt` (or complex numbers of those types). Random floating point numbers are generated uniformly in $[0, 1)$. As `BigInt` represents unbounded integers, the interval must be specified (e.g. `rand(big.(1:6))`).

Additionally, normal and exponential distributions are implemented for some `AbstractFloat` and `Complex` types, see `randn` and `randexp` for details.

To generate random numbers from other distributions, see the Distributions.jl package.

Warning

Because the precise way in which random numbers are generated is considered an implementation detail, bug fixes and speed improvements may change the stream of numbers that are generated after a version change. Relying on a specific seed or generated stream of numbers during unit testing is thus discouraged - consider testing properties of the methods in question instead.

91.1 Random numbers module

`Random.Random` – Module.

```
Random
```

Support for generating random numbers. Provides `rand`, `randn`, `AbstractRNG`, `Xoshiro`, `MersenneTwister`, and `RandomDevice`.

91.2 Random generation functions

`Base.rand` – Function.

```
rand([rng=default_rng()], [S], [dims...])
```

Pick a random element or array of random elements from the set of values specified by S; S can be

- an indexable collection (for example `1:9` or `('x', "y", :z)`)
- an `AbstractDict` or `AbstractSet` object
- a string (considered as a collection of characters), or
- a type from the list below, corresponding to the specified set of values
  - concrete integer types sample from `typemin(S):typemax(S)` (excepting `BigInt` which is not supported)
  - concrete floating point types sample from `[0, 1)`
  - concrete complex types `Complex{T}` if T is a sampleable type take their real and imaginary components independently from the set of values corresponding to T, but are not supported if T is not sampleable.
  - all `<:AbstractChar` types sample from the set of valid Unicode scalars
  - a user-defined type and set of values; for implementation guidance please see Hooking into the Random API
  - a tuple type of known size and where each parameter of S is itself a sampleable type; return a value of type S. Note that tuple types such as `Tuple{Vararg{T}}` (unknown size) and `Tuple{1:2}` (parameterized with a value) are not supported
  - a `Pair` type, e.g. `Pair{X, Y}` such that `rand` is defined for X and Y, in which case random pairs are produced.

S defaults to `Float64`. When only one argument is passed besides the optional `rng` and is a `Tuple`, it is interpreted as a collection of values (S) and not as `dims`.

See also `randn` for normally distributed numbers, and `rand!` and `randn!` for the in-place equivalents.

Julia 1.1

Support for S as a tuple requires at least Julia 1.1.

Julia 1.11

Support for S as a `Tuple` type requires at least Julia 1.11.

Examples

```
julia> rand(Int, 2)
2-element Vector{Int64}:
 1339893410598768192
 1575814717733606317

julia> using Random

julia> rand(Xoshiro(0), Dict(1=>2, 3=>4))
3 => 4

julia> rand((2, 3))
3

julia> rand(Float64, (2, 3))
2×3 Matrix{Float64}:
 0.999717  0.0143835  0.540787
 0.696556  0.783855   0.938235
```

Note

The complexity of `rand(rng, s::Union{AbstractDict,AbstractSet})` is linear in the length of `s`, unless an optimized method with constant complexity is available, which is the case for `Dict`, `Set` and dense `BitSet`s. For more than a few calls, use `rand(rng, collect(s))` instead, or either `rand(rng, Dict(s))` or `rand(rng, Set(s))` as appropriate.

`Random.rand!` – Function.

```
rand!([rng=default_rng()], A, [S=eltype(A)])
```

Populate the array `A` with random values. If `S` is specified (`S` can be a type or a collection, cf. `rand` for details), the values are picked randomly from `S`. This is equivalent to `copyto!(A, rand(rng, S, size(A)))` but without allocating a new array.

Examples

```
julia> rand!(Xoshiro(123), zeros(5))
5-element Vector{Float64}:
 0.521213795535383
 0.5868067574533484
 0.8908786980927811
 0.19090669902576285
 0.5256623915420473
```

Random.bitrand – Function.

```
bitrand([rng=default_rng()], [dims...])
```

Generate a BitArray of random boolean values.

Examples

```
julia> bitrand(Xoshiro(123), 10)
10-element BitVector:
 0
 1
 0
 1
 0
 1
 0
 0
 1
 1
```

Base.randn – Function.

```
randn([rng=default_rng()], [T=Float64], [dims...])
```

Generate a normally-distributed random number of type T with mean 0 and standard deviation 1. Given the optional dims argument(s), generate an array of size dims of such numbers. Julia's standard library supports randn for any floating-point type that implements rand, e.g. the Base types Float16, Float32, Float64 (the default), and BigFloat, along with their Complex counterparts.

(When T is complex, the values are drawn from the circularly symmetric complex normal distribution of variance 1, corresponding to real and imaginary parts having independent normal distribution with mean zero and variance 1/2).

See also randn! to act in-place.

Examples

Generating a single random number (with the default Float64 type):

```
julia> randn()
-0.942481877315864
```

Generating a matrix of normal random numbers (with the default Float64 type):

```
julia> randn(2,3)
2×3 Matrix{Float64}:
  1.18786   -0.678616   1.49463
 -0.342792  -0.134299  -1.45005
```

Setting up of the random number generator rng with a user-defined seed (for reproducible numbers) and using it to generate a random Float32 number or a matrix of ComplexF32 random numbers:

```
julia> using Random

julia> rng = Xoshiro(123);

julia> randn(rng, Float32)
-0.6457307f0

julia> randn(rng, ComplexF32, (2, 3))
2×3 Matrix{ComplexF32}:
  -1.03467-1.14806im  0.693657+0.056538im   0.291442+0.419454im
 -0.153912+0.34807im    1.0954-0.948661im  -0.543347-0.0538589im
```

Random.randn! – Function.

```
randn!([rng=default_rng()], A::AbstractArray) -> A
```

Fill the array A with normally-distributed (mean 0, standard deviation 1) random numbers. Also see the rand function.

Examples

```
julia> randn!(Xoshiro(123), zeros(5))
5-element Vector{Float64}:
 -0.6457306721039767
 -1.4632513788889214
 -1.6236037455860806
 -0.21766510678354617
  0.4922456865251828
```

Random.randexp – Function.

```
randexp([rng=default_rng()], [T=Float64], [dims...])
```

Generate a random number of type T according to the exponential distribution with scale 1. Optionally generate an array of such random numbers. The Base module currently provides an implementation for the types Float16, Float32, and Float64 (the default).

Examples

```
julia> rng = Xoshiro(123);

julia> randexp(rng, Float32)
1.1757717f0

julia> randexp(rng, 3, 3)
3×3 Matrix{Float64}:
 1.37766  0.456653  0.236418
 3.40007  0.229917  0.0684921
 0.48096  0.577481  0.71835
```

Random.randexp! – Function.

```
randexp!([rng=default_rng()], A::AbstractArray) -> A
```

Fill the array A with random numbers following the exponential distribution (with scale 1).

Examples

```
julia> randexp!(Xoshiro(123), zeros(5))
5-element Vector{Float64}:
 1.1757716836348473
 1.758884569451514
 1.0083623637301151
 0.3510644315565272
 0.6348266443720407
```

Random.randstring – Function.

```
randstring([rng=default_rng()], [chars], [len=8])
```

Create a random string of length len, consisting of characters from chars, which defaults to the set of upper- and lower-case letters and the digits 0-9. The optional rng argument specifies a random number generator, see Random Numbers.

Examples

```
julia> Random.seed!(3); randstring()
"Lxz5hUwn"

julia> randstring(Xoshiro(3), 'a':'z', 6)
"iyzcsm"

julia> randstring("ACGT")
"TGCTCCTC"
```

Note

chars can be any collection of characters, of type Char or UInt8 (more efficient), provided rand can randomly pick characters from it.

91.3 Subsequences, permutations and shuffling

Random.randsubseq – Function.

```
randsubseq([rng=default_rng(),] A, p) -> Vector
```

Return a vector consisting of a random subsequence of the given array A, where each element of A is included (in order) with independent probability p. (Complexity is linear in p*length(A), so this function is efficient even if p is small and A is large.) Technically, this process is known as "Bernoulli sampling" of A.

Examples

```
julia> randsubseq(Xoshiro(123), 1:8, 0.3)
2-element Vector{Int64}:
 4
 7
```

Random.randsubseq! – Function.

```
randsubseq!([rng=default_rng(),] S, A, p)
```

Like randsubseq, but the results are stored in S (which is resized as needed).

Examples

```
julia> S = Int64[];

julia> randsubseq!(Xoshiro(123), S, 1:8, 0.3)
2-element Vector{Int64}:
 4
 7

julia> S
2-element Vector{Int64}:
 4
 7
```

Random.randperm – Function.

```
randperm([rng=default_rng(),] n::Integer)
```

Construct a random permutation of length n. The optional rng argument specifies a random number generator (see Random Numbers). The element type of the result is the same as the type of n.

To randomly permute an arbitrary vector, see shuffle or shuffle!.

Julia 1.1

In Julia 1.1 randperm returns a vector v with eltype(v) == typeof(n) while in Julia 1.0 eltype(v) == Int.

Examples

```
julia> randperm(Xoshiro(0), 6)
6-element Vector{Int64}:
 5
 1
 2
 6
 3
 4
```

Random.randperm! – Function.

```
randperm!([rng=default_rng(),] A::AbstractArray{<:Integer})
```

Construct in A a random permutation of length length(A). The optional rng argument specifies a random number generator (see Random Numbers). To randomly permute an arbitrary vector, see shuffle or shuffle!.

Julia 1.13

`A isa Array` was required prior to Julia v1.13.

Examples

```
julia> randperm!(Xoshiro(0), Vector{Int}(undef, 6))
6-element Vector{Int64}:
 5
 1
 2
 6
 3
 4
```

`Random.randcycle` – Function.

```
randcycle([rng=default_rng(),] n::Integer)
```

Construct a random cyclic permutation of length `n`. The optional `rng` argument specifies a random number generator, see Random Numbers. The element type of the result is the same as the type of `n`.

Here, a "cyclic permutation" means that all of the elements lie within a single cycle. If `n > 0`, there are $(n-1)!$ possible cyclic permutations, which are sampled uniformly. If `n == 0`, `randcycle` returns an empty vector.

`randcycle!` is an in-place variant of this function.

Julia 1.1

In Julia 1.1 and above, `randcycle` returns a vector `v` with `eltype(v) == typeof(n)` while in Julia 1.0 `eltype(v) == Int`.

Examples

```
julia> randcycle(Xoshiro(0), 6)
6-element Vector{Int64}:
 5
 1
 4
 6
 3
 2
```

`Random.randcycle!` – Function.

```
randcycle!([rng=default_rng(),] A::AbstractArray{<:Integer})
```

Construct in `A` a random cyclic permutation of length `n = length(A)`. The optional `rng` argument specifies a random number generator, see Random Numbers.

Here, a "cyclic permutation" means that all of the elements lie within a single cycle. If `A` is nonempty (`n > 0`), there are $(n-1)!$ possible cyclic permutations, which are sampled uniformly. If `A` is empty, `randcycle!` leaves it unchanged.

`randcycle` is a variant of this function that allocates a new vector.

Julia 1.13

`A isa Array` was required prior to Julia v1.13.

Examples

```
julia> randcycle!(Xoshiro(0), Vector{Int}(undef, 6))
6-element Vector{Int64}:
 5
 1
 4
 6
 3
 2
```

`Random.shuffle` – Function.

```
shuffle([rng=default_rng(),] v::Union{NTuple,AbstractArray})
```

Return a randomly permuted copy of `v`. The optional `rng` argument specifies a random number generator (see Random Numbers). To permute `v` in-place, see `shuffle!`. To obtain randomly permuted indices, see `randperm`.

Julia 1.13

Shuffling an `NTuple` value requires Julia v1.13 or above.

Examples

```
julia> shuffle(Xoshiro(0), 1:6)
6-element Vector{Int64}:
 5
 1
 2
 6
 3
 4
```

`Random.shuffle!` – Function.

```
shuffle!([rng=default_rng(),] v::AbstractArray)
```

In-place version of `shuffle`: randomly permute `v` in-place, optionally supplying the random-number generator `rng`.

Examples

```
julia> shuffle!(Xoshiro(0), Vector(1:6))
6-element Vector{Int64}:
 5
 1
```

```
2
6
3
4
```

91.4 Generators (creation and seeding)

`Random.default_rng` – Function.

```
Random.default_rng() -> rng
```

Return the default global random number generator (RNG), which is used by `rand`-related functions when no explicit RNG is provided.

When the `Random` module is loaded, the default RNG is *randomly* seeded, via `Random.seed!()`: this means that each time a new julia session is started, the first call to `rand()` produces a different result, unless `seed!(seed)` is called first.

It is thread-safe: distinct threads can safely call `rand`-related functions on `default_rng()` concurrently, e.g. `rand(default_rng())`.

Note

The type of the default RNG is an implementation detail. Across different versions of Julia, you should not expect the default RNG to always have the same type, nor that it will produce the same stream of random numbers for a given seed.

Julia 1.3

This function was introduced in Julia 1.3.

`Random.seed!` – Function.

```
seed!([rng=default_rng()], seed) -> rng
seed!([rng=default_rng()]) -> rng
```

Reseed the random number generator: `rng` will give a reproducible sequence of numbers if and only if a `seed` is provided. Some RNGs don't accept a seed, like `RandomDevice`. After the call to `seed!`, `rng` is equivalent to a newly created object initialized with the same seed.

The types of accepted seeds depend on the type of `rng`, but in general, integer seeds should work. Providing `nothing` as the seed should be equivalent to not providing one.

If `rng` is not specified, it defaults to seeding the state of the shared task-local generator.

Examples

```
julia> Random.seed!(1234);

julia> x1 = rand(2)
2-element Vector{Float64}:
 0.32597672886359486
 0.5490511363155669
```

```
julia> Random.seed!(1234);

julia> x2 = rand(2)
2-element Vector{Float64}:
 0.32597672886359486
 0.5490511363155669

julia> x1 == x2
true

julia> rng = Xoshiro(1234); rand(rng, 2) == x1
true

julia> Xoshiro(1) == Random.seed!(rng, 1)
true

julia> rand(Random.seed!(rng), Bool) # not reproducible
true

julia> rand(Random.seed!(rng), Bool) # not reproducible either
false

julia> rand(Xoshiro(), Bool) # not reproducible either
true
```

`Random.AbstractRNG` – Type.

```
AbstractRNG
```

Supertype for random number generators such as `MersenneTwister` and `RandomDevice`.

`Random.TaskLocalRNG` – Type.

```
TaskLocalRNG
```

The `TaskLocalRNG` has state that is local to its task, not its thread. It is seeded upon task creation, from the state of its parent task, but without advancing the state of the parent's RNG.

As an upside, the `TaskLocalRNG` is pretty fast, and permits reproducible multithreaded simulations (barring race conditions), independent of scheduler decisions. As long as the number of threads is not used to make decisions on task creation, simulation results are also independent of the number of available threads / CPUs. The random stream should not depend on hardware specifics, up to endianness and possibly word size.

When seeding `TaskLocalRNG()` with `seed!`, the passed seed, if any, may be any integer.

Julia 1.11

Seeding `TaskLocalRNG()` with a negative integer seed requires at least Julia 1.11.

Julia 1.10

Task creation no longer advances the parent task's RNG state as of Julia 1.10.

`Random.Xoshiro` – Type.

```
Xoshiro(seed::Union{Integer, AbstractString})
Xoshiro()
```

Xoshiro256++ is a fast pseudorandom number generator described by David Blackman and Sebastiano Vigna in "Scrambled Linear Pseudorandom Number Generators", ACM Trans. Math. Softw., 2021. Reference implementation is available at https://prng.di.unimi.it

Apart from the high speed, Xoshiro has a small memory footprint, making it suitable for applications where many different random states need to be held for long time.

Julia's Xoshiro implementation has a bulk-generation mode; this seeds new virtual PRNGs from the parent, and uses SIMD to generate in parallel (i.e. the bulk stream consists of multiple interleaved xoshiro instances). The virtual PRNGs are discarded once the bulk request has been serviced (and should cause no heap allocations).

If no seed is provided, a randomly generated one is created (using entropy from the system). See the `seed!` function for reseeding an already existing `Xoshiro` object.

Julia 1.11

Passing a negative integer seed requires at least Julia 1.11.

Examples

```
julia> using Random

julia> rng = Xoshiro(1234);

julia> x1 = rand(rng, 2)
2-element Vector{Float64}:
 0.32597672886359486
 0.5490511363155669

julia> rng = Xoshiro(1234);

julia> x2 = rand(rng, 2)
2-element Vector{Float64}:
 0.32597672886359486
 0.5490511363155669

julia> x1 == x2
true
```

`Random.MersenneTwister` – Type.

```
MersenneTwister(seed)
MersenneTwister()
```

Create a `MersenneTwister` RNG object. Different RNG objects can have their own seeds, which may be useful for generating different streams of random numbers. The seed may be an integer, a string, or a vector of `UInt32` integers. If no seed is provided, a randomly generated one is created (using entropy from the system). See the `seed!` function for reseeding an already existing `MersenneTwister` object.

Julia 1.11

Passing a negative integer seed requires at least Julia 1.11.

Examples

```
julia> rng = MersenneTwister(123);

julia> x1 = rand(rng, 2)
2-element Vector{Float64}:
 0.37453777969575874
 0.8735343642013971

julia> x2 = rand(MersenneTwister(123), 2)
2-element Vector{Float64}:
 0.37453777969575874
 0.8735343642013971

julia> x1 == x2
true
```

`Random.RandomDevice` – Type.

```
RandomDevice()
```

Create a `RandomDevice` RNG object. Two such objects will always generate different streams of random numbers. The entropy is obtained from the operating system.

91.5 Hooking into the `Random` API

There are two mostly orthogonal ways to extend `Random` functionalities:

1. generating random values of custom types
2. creating new generators

The API for 1) is quite functional, but is relatively recent so it may still have to evolve in subsequent releases of the `Random` module. For example, it's typically sufficient to implement one `rand` method in order to have all other usual methods work automatically.

The API for 2) is still rudimentary, and may require more work than strictly necessary from the implementer, in order to support usual types of generated values.

Generating random values of custom types

Generating random values for some distributions may involve various trade-offs. *Pre-computed* values, such as an alias table for discrete distributions, or "squeezing" functions for univariate distributions, can speed up sampling considerably. How much information should be pre-computed can depend on the number of values we plan to draw from a distribution. Also, some random number generators can have certain properties that various algorithms may want to exploit.

The Random module defines a customizable framework for obtaining random values that can address these issues. Each invocation of rand generates a *sampler* which can be customized with the above trade-offs in mind, by adding methods to Sampler, which in turn can dispatch on the random number generator, the object that characterizes the distribution, and a suggestion for the number of repetitions. Currently, for the latter, Val{1} (for a single sample) and Val{Inf} (for an arbitrary number) are used, with Random.Repetition an alias for both.

The object returned by Sampler is then used to generate the random values. When implementing the random generation interface for a value X that can be sampled from, the implementer should define the method

```
rand(rng, sampler)
```

for the particular sampler returned by Sampler(rng, X, repetition).

Samplers can be arbitrary values that implement rand(rng, sampler), but for most applications the following predefined samplers may be sufficient:

1. SamplerType{T}() can be used for implementing samplers that draw from type T (e.g. rand(Int)). This is the default returned by Sampler for *types*.
2. SamplerTrivial(self) is a simple wrapper for self, which can be accessed with []. This is the recommended sampler when no pre-computed information is needed (e.g. rand(1:3)), and is the default returned by Sampler for *values*.
3. SamplerSimple(self, data) also contains the additional data field, which can be used to store arbitrary pre-computed values, which should be computed in a *custom method* of Sampler.

We provide examples for each of these. We assume here that the choice of algorithm is independent of the RNG, so we use AbstractRNG in our signatures.

Random.Sampler – Type.

```
Sampler(rng, x, repetition = Val(Inf))
```

Return a sampler object that can be used to generate random values from rng for x.

When sp = Sampler(rng, x, repetition), rand(rng, sp) will be used to draw random values, and should be defined accordingly.

repetition can be Val(1) or Val(Inf), and should be used as a suggestion for deciding the amount of precomputation, if applicable.

Random.SamplerType and Random.SamplerTrivial are default fallbacks for *types* and *values*, respectively. Random.SamplerSimple can be used to store pre-computed values without defining extra types for only this purpose.

Random.SamplerType – Type.

```
SamplerType{T}()
```

A sampler for types, containing no other information. The default fallback for Sampler when called with types.

`Random.SamplerTrivial` – Type.

```
SamplerTrivial(x)
```

Create a sampler that just wraps the given value `x`. This is the default fall-back for values. The `eltype` of this sampler is equal to `eltype(x)`.

The recommended use case is sampling from values without precomputed data.

`Random.SamplerSimple` – Type.

```
SamplerSimple(x, data)
```

Create a sampler that wraps the given value `x` and the `data`. The `eltype` of this sampler is equal to `eltype(x)`.

The recommended use case is sampling from values with precomputed data.

Decoupling pre-computation from actually generating the values is part of the API, and is also available to the user. As an example, assume that `rand(rng, 1:20)` has to be called repeatedly in a loop: the way to take advantage of this decoupling is as follows:

```
rng = Xoshiro()
sp = Random.Sampler(rng, 1:20) # or Random.Sampler(Xoshiro, 1:20)
for x in X
    n = rand(rng, sp) # similar to n = rand(rng, 1:20)
    # use n
end
```

This is the mechanism that is also used in the standard library, e.g. by the default implementation of random array generation (like in `rand(1:20, 10)`).

Generating values from a type

Given a type `T`, it's currently assumed that if `rand(T)` is defined, an object of type `T` will be produced. `SamplerType` is the *default sampler for types*. In order to define random generation of values of type `T`, the `rand(rng::AbstractRNG, ::Random.SamplerType{T})` method should be defined, and should return values what `rand(rng, T)` is expected to return.

Let's take the following example: we implement a `Die` type, with a variable number `n` of sides, numbered from 1 to `n`. We want `rand(Die)` to produce a `Die` with a random number of up to 20 sides (and at least 4):

```
struct Die
    nsides::Int # number of sides
end

Random.rand(rng::AbstractRNG, ::Random.SamplerType{Die}) = Die(rand(rng, 4:20))

# output
```

Scalar and array methods for `Die` now work as expected:

```
julia> rand(Die)
Die(5)

julia> rand(Xoshiro(0), Die)
Die(10)

julia> rand(Die, 3)
3-element Vector{Die}:
 Die(9)
 Die(15)
 Die(14)

julia> a = Vector{Die}(undef, 3); rand!(a)
3-element Vector{Die}:
 Die(19)
 Die(7)
 Die(17)
```

A simple sampler without pre-computed data

Here we define a sampler for a collection. If no pre-computed data is required, it can be implemented with a `SamplerTrivial` sampler, which is in fact the *default fallback for values*.

In order to define random generation out of objects of type `S`, the following method should be defined: `rand(rng::AbstractRNG, sp::Random.SamplerTrivial{S})`. Here, `sp` simply wraps an object of type `S`, which can be accessed via `sp[]`. Continuing the `Die` example, we want now to define `rand(d::Die)` to produce an `Int` corresponding to one of `d`'s sides:

```
julia> Random.rand(rng::AbstractRNG, d::Random.SamplerTrivial{Die}) = rand(rng, 1:d[].nsides);

julia> rand(Die(4))
1

julia> rand(Die(4), 3)
3-element Vector{Any}:
 2
 3
 3
```

Given a collection type `S`, it's currently assumed that if `rand(::S)` is defined, an object of type `eltype(S)` will be produced. In the last example, a `Vector{Any}` is produced; the reason is that `eltype(Die) == Any`. The remedy is to define `Base.eltype(::Type{Die}) = Int`.

Generating values for an `AbstractFloat` type

`AbstractFloat` types are special-cased, because by default random values are not produced in the whole type domain, but rather in `[0,1)`. The following method should be implemented for `T <: AbstractFloat`: `Random.rand(::AbstractRNG, ::Random.SamplerTrivial{Random.CloseOpen01{T}})`

An optimized sampler with pre-computed data

Consider a discrete distribution, where numbers `1:n` are drawn with given probabilities that sum to one. When many values are needed from this distribution, the fastest method is using an alias table. We don't

provide the algorithm for building such a table here, but suppose it is available in `make_alias_table(probabilities)` instead, and `draw_number(rng, alias_table)` can be used to draw a random number from it.

Suppose that the distribution is described by

```
struct DiscreteDistribution{V <: AbstractVector}
    probabilities::V
end
```

and that we *always* want to build an alias table, regardless of the number of values needed (we learn how to customize this below). The methods

```
Random.eltype(::Type{<:DiscreteDistribution}) = Int

function Random.Sampler(::Type{<:AbstractRNG}, distribution::DiscreteDistribution, ::Repetition)
    SamplerSimple(distribution, make_alias_table(distribution.probabilities))
end
```

should be defined to return a sampler with pre-computed data, then

```
function rand(rng::AbstractRNG, sp::SamplerSimple{<:DiscreteDistribution})
    draw_number(rng, sp.data)
end
```

will be used to draw the values.

Custom sampler types

The `SamplerSimple` type is sufficient for most use cases with precomputed data. However, in order to demonstrate how to use custom sampler types, here we implement something similar to `SamplerSimple`.

Going back to our `Die` example: `rand(::Die)` uses random generation from a range, so there is an opportunity for this optimization. We call our custom sampler `SamplerDie`.

```
import Random: Sampler, rand

struct SamplerDie <: Sampler{Int} # generates values of type Int
    die::Die
    sp::Sampler{Int} # this is an abstract type, so this could be improved
end

Sampler(RNG::Type{<:AbstractRNG}, die::Die, r::Random.Repetition) =
    SamplerDie(die, Sampler(RNG, 1:die.nsides, r))
# the `r` parameter will be explained later on

rand(rng::AbstractRNG, sp::SamplerDie) = rand(rng, sp.sp)
```

It's now possible to get a sampler with `sp = Sampler(rng, die)`, and use `sp` instead of `die` in any `rand` call involving `rng`. In the simplistic example above, `die` doesn't need to be stored in `SamplerDie` but this is often the case in practice.

Of course, this pattern is so frequent that the helper type used above, namely `Random.SamplerSimple`, is available, saving us the definition of `SamplerDie`: we could have implemented our decoupling with:

```
Sampler(RNG::Type{<:AbstractRNG}, die::Die, r::Random.Repetition) =
    SamplerSimple(die, Sampler(RNG, 1:die.nsides, r))

rand(rng::AbstractRNG, sp::SamplerSimple{Die}) = rand(rng, sp.data)
```

Here, `sp.data` refers to the second parameter in the call to the `SamplerSimple` constructor (in this case equal to `Sampler(rng, 1:die.nsides, r)`), while the `Die` object can be accessed via `sp[]`.

Like `SamplerDie`, any custom sampler must be a subtype of `Sampler{T}` where T is the type of the generated values. Note that `SamplerSimple(x, data) isa Sampler{eltype(x)}`, so this constrains what the first argument to `SamplerSimple` can be (it's recommended to use `SamplerSimple` like in the `Die` example, where `x` is simply forwarded while defining a `Sampler` method). Similarly, `SamplerTrivial(x) isa Sampler{eltype(x)}`.

Another helper type is currently available for other cases, `Random.SamplerTag`, but is considered as internal API, and can break at any time without proper deprecations.

Using distinct algorithms for scalar or array generation

In some cases, whether one wants to generate only a handful of values or a large number of values will have an impact on the choice of algorithm. This is handled with the third parameter of the `Sampler` constructor. Let's assume we defined two helper types for `Die`, say `SamplerDie1` which should be used to generate only few random values, and `SamplerDieMany` for many values. We can use those types as follows:

```
Sampler(RNG::Type{<:AbstractRNG}, die::Die, ::Val{1}) = SamplerDie1(...)
Sampler(RNG::Type{<:AbstractRNG}, die::Die, ::Val{Inf}) = SamplerDieMany(...)
```

Of course, `rand` must also be defined on those types (i.e. `rand(::AbstractRNG, ::SamplerDie1)` and `rand(::AbstractRNG, ::SamplerDieMany)`). Note that, as usual, `SamplerTrivial` and `SamplerSimple` can be used if custom types are not necessary.

Note: `Sampler(rng, x)` is simply a shorthand for `Sampler(rng, x, Val(Inf))`, and `Random.Repetition` is an alias for `Union{Val{1}, Val{Inf}}`.

Creating new generators

The API is not clearly defined yet, but as a rule of thumb:

1. any `rand` method producing "basic" types (`isbitstype` integer and floating types in `Base`) should be defined for this specific RNG, if they are needed;
2. other documented `rand` methods accepting an `AbstractRNG` should work out of the box, (provided the methods from 1) what are relied on are implemented), but can of course be specialized for this RNG if there is room for optimization;
3. `copy` for pseudo-RNGs should return an independent copy that generates the exact same random sequence as the original from that point when called in the same way. When this is not feasible (e.g. hardware-based RNGs), `copy` must not be implemented.

Concerning 1), a `rand` method may happen to work automatically, but it's not officially supported and may break without warnings in a subsequent release.

To define a new `rand` method for an hypothetical `MyRNG` generator, and a value specification `s` (e.g. `s == Int`, or `s == 1:10`) of type `S==typeof(s)` or `S==Type{s}` if `s` is a type, the same two methods as we saw before must be defined:

1. `Sampler(::Type{MyRNG}, ::S, ::Repetition)`, which returns an object of type say `SamplerS`
2. `rand(rng::MyRNG, sp::SamplerS)`

It can happen that `Sampler(rng::AbstractRNG, ::S, ::Repetition)` is already defined in the `Random` module. It would then be possible to skip step 1) in practice (if one wants to specialize generation for this particular RNG type), but the corresponding `SamplerS` type is considered as internal detail, and may be changed without warning.

Specializing array generation

In some cases, for a given RNG type, generating an array of random values can be more efficient with a specialized method than by merely using the decoupling technique explained before. This is for example the case for `MersenneTwister`, which natively writes random values in an array.

To implement this specialization for `MyRNG` and for a specification `s`, producing elements of type `S`, the following method can be defined: `rand!(rng::MyRNG, a::AbstractArray{S}, ::SamplerS)`, where `SamplerS` is the type of the sampler returned by `Sampler(MyRNG, s, Val(Inf))`. Instead of `AbstractArray`, it's possible to implement the functionality only for a subtype, e.g. `Array{S}`. The non-mutating array method of `rand` will automatically call this specialization internally.

Chapter 92

Reproducibility

By using an RNG parameter initialized with a given seed, you can reproduce the same pseudorandom number sequence when running your program multiple times. However, a minor release of Julia (e.g. 1.3 to 1.4) *may change* the sequence of pseudorandom numbers generated from a specific seed. (Even if the sequence produced by a low-level function like `rand` does not change, the output of higher-level functions like `randsubseq` may change due to algorithm updates.) Rationale: guaranteeing that pseudorandom streams never change prohibits many algorithmic improvements.

If you need to guarantee exact reproducibility of random data, it is advisable to simply *save the data* (e.g. as a supplementary attachment in a scientific publication). (You can also, of course, specify a particular Julia version and package manifest, especially if you require bit reproducibility.)

Software tests that rely on *specific* "random" data should also generally either save the data, embed it into the test code, or use third-party packages like StableRNGs.jl. On the other hand, tests that should pass for *most* random data (e.g. testing `A \ (A*x) ≈ x` for a random matrix `A = randn(n,n)`) can use an RNG with a fixed seed to ensure that simply running the test many times does not encounter a failure due to very improbable data (e.g. an extremely ill-conditioned matrix).

The statistical *distribution* from which random samples are drawn *is* guaranteed to be the same across any minor Julia releases.

Chapter 93

The Julia REPL

Julia comes with a full-featured interactive command-line REPL (read-eval-print loop) built into the `julia` executable. In addition to allowing quick and easy evaluation of Julia statements, it has a searchable history, tab-completion, many helpful keybindings, and dedicated help and shell modes. The REPL can be started by simply calling `julia` with no arguments or double-clicking on the executable:

```
$ julia


               _
   _       _ _(_)_     |  Documentation: https://docs.julialang.org
  (_)     | (_) (_)    |
   _ _   _| |_  __ _   |  Type "?" for help, "]?" for Pkg help.
  | | | | | | |/ _` |  |
  | | |_| | | | (_| |  |  Version 1.13.0-rc4 (2026-09-02)
 _/ |\__'_|_|_|\__'_|  |  Official https://julialang.org release
|__/                   |


julia>
```

To exit the interactive session, type `^D` – the control key together with the `d` key on a blank line – or type `exit()` followed by the return or enter key. The REPL greets you with a banner and a `julia>` prompt.

93.1 The different prompt modes

The Julian mode

The REPL has five main modes of operation. The first and most common is the Julian prompt. It is the default mode of operation; each new line initially starts with `julia>`. It is here that you can enter Julia expressions. Hitting return or enter after a complete expression has been entered will evaluate the entry and show the result of the last expression.

```
julia> string(1 + 2)
"3"
```

There are a number of useful features unique to interactive work. In addition to showing the result, the REPL also binds the result to the variable ans. A trailing semicolon on the line can be used as a flag to suppress showing the result.

```
julia> string(3 * 4);

julia> ans
"12"
```

In Julia mode, the REPL supports something called *prompt pasting*. This activates when pasting text that starts with julia> into the REPL. In that case, only expressions starting with julia> (as well as the other REPL mode prompts: shell>, help?>, pkg>) are parsed, but others are removed. This makes it possible to paste a chunk of text that has been copied from a REPL session without having to scrub away prompts and outputs. This feature is enabled by default but can be disabled or enabled at will with REPL.enable_promptpaste(::Bool). If it is enabled, you can try it out by pasting the code block above this paragraph straight into the REPL. This feature does not work on the standard Windows command prompt due to its limitation at detecting when a paste occurs.

A non-nothing result of executing an expression is displayed by the REPL using the show function with a specific IOContext (via display, which defaults to calling show(io, MIME("text/plain"), ans), which in turn defaults to show(io, ans)). In particular, the :limit attribute is set to true. Other attributes can receive in certain show methods a default value if it's not already set, like :compact. It's possible, as an experimental feature, to specify the attributes used by the REPL via the Base.active_repl.options.iocontext dictionary (associating values to attributes). For example:

```
julia> rand(2, 2)
2×2 Matrix{Float64}:
 0.8833    0.329197
 0.719708  0.59114

julia> show(IOContext(stdout, :compact => false), "text/plain", rand(2, 2))
 0.43540323669187075  0.15759787870609387
 0.2540832269192739   0.4597637838786053
julia> Base.active_repl.options.iocontext[:compact] = false;

julia> rand(2, 2)
2×2 Matrix{Float64}:
 0.2083967319174056  0.13330606013126012
 0.6244375177790158  0.9777957560761545
```

In order to define automatically the values of this dictionary at startup time, one can use the atreplinit function in the ~/.julia/config/startup.jl file, for example:

```
atreplinit() do repl
    repl.options.iocontext[:compact] = false
end
```

Help mode

When the cursor is at the beginning of the line, the prompt can be changed to a help mode by typing ?. Julia will attempt to print help or documentation for anything entered in help mode:

```
julia> ? # upon typing ?, the prompt changes (in place) to: help?>

help?> string
search: string String Cstring Cwstring RevString randstring bytestring SubString

  string(xs...)

  Create a string from any values using the print function.
```

Macros, types and variables can also be queried:

```
help?> @time
  @time

  A macro to execute an expression, printing the time it took to execute, the number of
  ↪  allocations,
  and the total number of bytes its execution caused to be allocated, before returning the value
  ↪  of the
  expression.

  See also @timev, @timed, @elapsed, and @allocated.

help?> Int32
search: Int32 UInt32

  Int32 <: Signed

  32-bit signed integer type.
```

A string or regex literal searches all docstrings using `apropos`:

```
help?> "aprop"
REPL.stripmd
Base.Docs.apropos

help?> r"ap..p"
Base.:∘
Base.shell_escape_posixly
Distributed.CachingPool
REPL.stripmd
Base.Docs.apropos
```

Another feature of help mode is the ability to access extended docstrings. You can do this by typing something like `??Print` rather than `?Print` which will display the `# Extended help` section from the source codes documentation.

Help mode can be exited by pressing backspace at the beginning of the line.

Shell mode

Just as help mode is useful for quick access to documentation, another common task is to use the system shell to execute system commands. Just as `?` entered help mode when at the beginning of the line, a

semicolon (;) will enter the shell mode. And it can be exited by pressing backspace at the beginning of the line.

```
julia> ; # upon typing ;, the prompt changes (in place) to: shell>

shell> echo hello
hello
```

Note

For Windows users, Julia's shell mode does not expose windows shell commands. Hence, this will fail:

```
julia> ; # upon typing ;, the prompt changes (in place) to: shell>

shell> dir
ERROR: IOError: could not spawn `dir`: no such file or directory (ENOENT)
Stacktrace!
.......
```

However, you can get access to `PowerShell` like this:

```
julia> ; # upon typing ;, the prompt changes (in place) to: shell>

shell> powershell
Windows PowerShell
Copyright (C) Microsoft Corporation. All rights reserved.
PS C:\Users\elm>
```

... and to `cmd.exe` like that (see the `dir` command):

```
julia> ; # upon typing ;, the prompt changes (in place) to: shell>

shell> cmd
Microsoft Windows [version 10.0.17763.973]
(c) 2018 Microsoft Corporation. All rights reserved.
C:\Users\elm>dir
 Volume in drive C has no label
 Volume Serial Number is 1643-0CD7
  Directory of C:\Users\elm

29/01/2020  22:15    <DIR>          .
29/01/2020  22:15    <DIR>          ..
02/02/2020  08:06    <DIR>          .atom
```

Pkg mode

The Package manager mode accepts specialized commands for loading and updating packages. It is entered by pressing the] key at the Julian REPL prompt and exited by pressing CTRL-C or pressing the backspace key at the beginning of the line. The prompt for this mode is pkg>. It supports its own help-mode, which is entered by pressing ? at the beginning of the line of the pkg> prompt. The Package manager mode is documented in the Pkg manual, available at https://julialang.github.io/Pkg.jl/v1/.

History searching

In all of the above modes, the executed lines get saved to a history file, which can be searched. To initiate an interactive search through the previous history, type ^R – the control key together with the r key.

You will be presented with an interactive history viewer. As you type your search history will be filtered; pressing enter will insert the selected history entry into the REPL. Detailed help for the history searcher is available within the REPL with the special queries ? and ??.

All executed commands in the Julia REPL are logged into ~/.julia/logs/repl_history.jl along with a timestamp of when it was executed and the current REPL mode you were in. The history searcher reads this log file in order to find the commands which you previously ran. Multiple REPLs can write to this file at once, and every time you begin a search the newest history is fetched. Use of this file can be disabled at startup by passing the --history-file=no flag to Julia.

93.2 Key bindings

The Julia REPL makes great use of key bindings. Several control-key bindings were already introduced above (^D to exit, ^R for searching), but there are many more. In addition to the control-key, there are also meta-key bindings. These vary more by platform, but most terminals default to using alt- or option- held down with a key to send the meta-key (or can be configured to do so), or pressing Esc and then the key.

Customizing keybindings

Julia's REPL keybindings may be fully customized to a user's preferences by passing a dictionary to REPL.setup_interface. The keys of this dictionary may be characters or strings. The key '*' refers to the default action. Control plus character x bindings are indicated with "^x". Meta plus x can be written "\\M-x" or "\ex", and Control plus x can be written "\\C-x" or "^x". The values of the custom keymap must be nothing (indicating that the input should be ignored) or functions that accept the signature (PromptState, AbstractREPL, Char). The REPL.setup_interface function must be called before the REPL is initialized, by registering the operation with atreplinit . For example, to bind the up and down arrow keys to move through history without prefix search, one could put the following code in ~/.julia/config/startup.jl:

```
import REPL
import REPL.LineEdit

const mykeys = Dict{Any,Any}(
    # Up Arrow
    "\e[A" => (s,o...)->(LineEdit.edit_move_up(s) || LineEdit.history_prev(s,
    ↪  LineEdit.mode(s).hist)),
    # Down Arrow
    "\e[B" => (s,o...)->(LineEdit.edit_move_down(s) || LineEdit.history_next(s,
    ↪  LineEdit.mode(s).hist))
)

function customize_keys(repl)
    repl.interface = REPL.setup_interface(repl; extra_repl_keymap = mykeys)
end

atreplinit(customize_keys)
```

Users should refer to LineEdit.jl to discover the available actions on key input.

Automatic bracket insertion

The Julia REPL supports automatically inserting closing brackets, parentheses, braces, and quotes when you type the opening character.

When enabled, typing an opening bracket `(`, `{`, or `[` will automatically insert the matching closing bracket `)`, `}`, or `]` and position the cursor between them. The same behavior applies to quotes (`"`, `'`, and `` ` ``). If you then type the closing character, the REPL will skip over the auto-inserted character instead of inserting a duplicate. Additionally, pressing backspace immediately after auto-insertion will remove both the opening and closing characters.

To disable this feature, add the following to your `~/.julia/config/startup.jl` file:

```
atreplinit() do repl
    # Robust against older julia versions
    if hasfield(typeof(repl.options), :auto_insert_closing_bracket)
        repl.options.auto_insert_closing_bracket = false
    end
end
```

93.3 Tab completion

In the Julian, pkg and help modes of the REPL, one can enter the first few characters of a function or type and then press the tab key to get a list all matches:

```
julia> x[TAB]
julia> xor
```

In some cases it only completes part of the name, up to the next ambiguity:

```
julia> mapf[TAB]
julia> mapfold
```

If you hit tab again, then you get the list of things that might complete this:

```
julia> mapfold[TAB]
mapfoldl mapfoldr
```

When a single complete tab-complete result is available at the end of an input line and 2 or more characters have been typed, a hint of the completion will show in a lighter color. This can be disabled via `Base.active_repl.options.hint_tab_completes = false` or by adding

```
atreplinit() do repl
    if VERSION >= v"1.11.0-0"
        repl.options.hint_tab_completes = false
    end
end
```

to your `~/.julia/config/startup.jl`.

Julia 1.11

Tab-complete hinting was added in Julia 1.11

Like other components of the REPL, the search is case-sensitive:

```
julia> stri[TAB]
stride     strides     string      strip

julia> Stri[TAB]
StridedArray    StridedMatrix    StridedVecOrMat  StridedVector    String
```

The tab key can also be used to substitute LaTeX math symbols with their Unicode equivalents, and get a list of LaTeX matches as well:

```
julia> \pi[TAB]
julia> π
π = 3.1415926535897...

julia> e\_1[TAB] = [1,0]
julia> e₁ = [1,0]
2-element Vector{Int64}:
 1
 0

julia> e\^1[TAB] = [1 0]
julia> e¹ = [1 0]
1×2 Matrix{Int64}:
 1  0

julia> \sqrt[TAB]2     # √ is equivalent to the sqrt function
julia> √2
1.4142135623730951

julia> \hbar[TAB](h) = h / 2\pi[TAB]
julia> ħ(h) = h / 2π
ħ (generic function with 1 method)

julia> \h[TAB]
\hat              \hermitconjmatrix  \hkswarow          \hrectangle
\hatapprox        \hexagon           \hookleftarrow     \hrectangleblack
\hbar             \hexagonblack      \hookrightarrow    \hslash
\heartsuit        \hksearow          \house             \hspace

julia> α="\alpha[TAB]"   # LaTeX completion also works in strings
julia> α="α"
```

A full list of tab-completions can be found in the Unicode Input section of the manual.

Completion of paths works for strings and julia's shell mode:

```
julia> path="/[TAB]"
.dockerenv  .juliabox/   boot/        etc/         lib/         media/       opt/         root/
↪  sbin/        sys/         usr/
.dockerinit bin/         dev/         home/        lib64/       mnt/         proc/        run/
↪  srv/         tmp/         var/
shell> /[TAB]
.dockerenv  .juliabox/   boot/        etc/         lib/         media/       opt/         root/
↪  sbin/        sys/         usr/
.dockerinit bin/         dev/         home/        lib64/       mnt/         proc/        run/
↪  srv/         tmp/         var/
```

Dictionary keys can also be tab completed:

```
julia> foo = Dict("qwer1"=>1, "qwer2"=>2, "asdf"=>3)
Dict{String,Int64} with 3 entries:
  "qwer2" => 2
  "asdf"  => 3
  "qwer1" => 1

julia> foo["q[TAB]

"qwer1" "qwer2"
julia> foo["qwer
```

Tab completion can also help completing fields:

```
julia> x = 3 + 4im;

julia> x.[TAB][TAB]
im re

julia> import UUIDs

julia> UUIDs.uuid[TAB][TAB]
uuid1        uuid4         uuid5        uuid_version
```

Fields for output from functions can also be completed:

```
julia> split("","")[1].[TAB]
lastindex  offset  string
```

The completion of fields for output from functions uses type inference, and it can only suggest fields if the function is type stable.

Tab completion can help with investigation of the available methods matching the input arguments:

```
julia> max([TAB] # All methods are displayed, not shown here due to size of the list

julia> max([1, 2], [TAB] # All methods where `Vector{Int}` matches as first argument
max(x, y) in Base at operators.jl:215
```

```
max(a, b, c, xs...) in Base at operators.jl:281

julia> max([1, 2], max(1, 2), [TAB] # All methods matching the arguments.
max(x, y) in Base at operators.jl:215
max(a, b, c, xs...) in Base at operators.jl:281
```

Keywords are also displayed in the suggested methods after `;`, see below line where `limit` and keepempty are keyword arguments:

```
julia> split("1 1 1", [TAB]
split(str::AbstractString; limit, keepempty) in Base at strings/util.jl:302
split(str::T, splitter; limit, keepempty) where T<:AbstractString in Base at strings/util.jl:277
```

The completion of the methods uses type inference and can therefore see if the arguments match even if the arguments are output from functions. The function needs to be type stable for the completion to be able to remove non-matching methods.

If you wonder which methods can be used with particular argument types, use `?` as the function name. This shows an example of looking for functions in InteractiveUtils that accept a single string:

```
julia> InteractiveUtils.?("somefile")[TAB]
edit(path::AbstractString) in InteractiveUtils at InteractiveUtils/src/editless.jl:197
less(file::AbstractString) in InteractiveUtils at InteractiveUtils/src/editless.jl:266
```

This listed methods in the `InteractiveUtils` module that can be called on a string. By default, this excludes methods where all arguments are typed as `Any`, but you can see those too by holding down SHIFT-TAB instead of TAB:

```
julia> InteractiveUtils.?("somefile")[SHIFT-TAB]
apropos(string) in REPL at REPL/src/docview.jl:796
clipboard(x) in InteractiveUtils at InteractiveUtils/src/clipboard.jl:64
code_llvm(f) in InteractiveUtils at InteractiveUtils/src/codeview.jl:221
code_native(f) in InteractiveUtils at InteractiveUtils/src/codeview.jl:243
edit(path::AbstractString) in InteractiveUtils at InteractiveUtils/src/editless.jl:197
edit(f) in InteractiveUtils at InteractiveUtils/src/editless.jl:225
eval(x) in InteractiveUtils at InteractiveUtils/src/InteractiveUtils.jl:3
include(x) in InteractiveUtils at InteractiveUtils/src/InteractiveUtils.jl:3
less(file::AbstractString) in InteractiveUtils at InteractiveUtils/src/editless.jl:266
less(f) in InteractiveUtils at InteractiveUtils/src/editless.jl:274
report_bug(kind) in InteractiveUtils at InteractiveUtils/src/InteractiveUtils.jl:391
separate_kwargs(args...; kwargs...) in InteractiveUtils at InteractiveUtils/src/macros.jl:7
```

You can also use `?("somefile")[TAB]` and look across all modules, but the method lists can be long.

By omitting the closing parenthesis, you can include functions that might require additional arguments:

```
julia> using Mmap

help?> Mmap.?("file",[TAB]
Mmap.Anonymous(name::String, readonly::Bool, create::Bool) in Mmap at Mmap/src/Mmap.jl:16
```

```
mmap(file::AbstractString) in Mmap at Mmap/src/Mmap.jl:245
mmap(file::AbstractString, ::Type{T}) where T<:Array in Mmap at Mmap/src/Mmap.jl:245
mmap(file::AbstractString, ::Type{T}, dims::Tuple{Vararg{Integer, N}}) where {T<:Array, N} in
↪  Mmap at Mmap/src/Mmap.jl:245
mmap(file::AbstractString, ::Type{T}, dims::Tuple{Vararg{Integer, N}}, offset::Integer; grow,
↪  shared) where {T<:Array, N} in Mmap at Mmap/src/Mmap.jl:245
mmap(file::AbstractString, ::Type{T}, len::Integer) where T<:Array in Mmap at
↪  Mmap/src/Mmap.jl:251
mmap(file::AbstractString, ::Type{T}, len::Integer, offset::Integer; grow, shared) where T<:Array
↪  in Mmap at Mmap/src/Mmap.jl:251
mmap(file::AbstractString, ::Type{T}, dims::Tuple{Vararg{Integer, N}}) where {T<:BitArray, N} in
↪  Mmap at Mmap/src/Mmap.jl:316
mmap(file::AbstractString, ::Type{T}, dims::Tuple{Vararg{Integer, N}}, offset::Integer; grow,
↪  shared) where {T<:BitArray, N} in Mmap at Mmap/src/Mmap.jl:316
mmap(file::AbstractString, ::Type{T}, len::Integer) where T<:BitArray in Mmap at
↪  Mmap/src/Mmap.jl:322
mmap(file::AbstractString, ::Type{T}, len::Integer, offset::Integer; grow, shared) where
↪  T<:BitArray in Mmap at Mmap/src/Mmap.jl:322
```

93.4 Syntax Highlighting

The REPL provides syntax highlighting for input as you type. Syntax highlighting is enabled by default but can be disabled in your `~/.julia/config/startup.jl`:

```
atreplinit() do repl
    repl.options.style_input = false
end
```

Customizing Syntax Highlighting Colors

The default syntax highlighting theme is quite conservative but can be customized using a TOML file `faces.toml` (https://julialang.github.io/StyledStrings.jl/dev/#stdlib-styledstrings-face-toml) in `.julia/config` (or by explicitly loading the faces from a face toml file).

<details> <summary>Example: Monokai color theme (click to expand)</summary>

```
# Monokai color theme for Julia syntax highlighting

[julia_macro]
foreground = "#A6E22E"

[julia_symbol]
foreground = "#AE81FF"

[julia_singleton_identifier]
inherit = "julia_symbol"

[julia_type]
foreground = "#66D9EF"

[julia_typedec]
foreground = "#66D9EF"
```

```
weight = "bold"

[julia_comment]
foreground = "#75715E"
italic = true

[julia_string]
foreground = "#E6DB74"

[julia_regex]
inherit = "julia_string"

[julia_backslash_literal]
foreground = "#FD971F"
inherit = "julia_string"

[julia_string_delim]
foreground = "#E6DB74"
weight = "bold"

[julia_cmdstring]
inherit = "julia_string"

[julia_char]
inherit = "julia_string"

[julia_char_delim]
inherit = "julia_string_delim"

[julia_number]
foreground = "#AE81FF"

[julia_bool]
foreground = "#AE81FF"
weight = "bold"

[julia_funcall]
foreground = "#A6E22E"

[julia_broadcast]
foreground = "#F92672"
weight = "bold"

[julia_builtin]
foreground = "#66D9EF"
weight = "bold"

[julia_operator]
foreground = "#F92672"

[julia_comparator]
inherit = "julia_operator"
```

```
[julia_assignment]
foreground = "#F92672"
weight = "bold"

[julia_keyword]
foreground = "#F92672"
weight = "bold"

[julia_parentheses]
foreground = "#F8F8F2"

[julia_unpaired_parentheses]
background = "#F92672"
foreground = "#F8F8F0"
weight = "bold"

[julia_error]
background = "#F92672"
foreground = "#F8F8F0"

[julia_rainbow_paren_1]
foreground = "#A6E22E"
inherit = "julia_parentheses"

[julia_rainbow_paren_2]
foreground = "#66D9EF"
inherit = "julia_parentheses"

[julia_rainbow_paren_3]
foreground = "#FD971F"
inherit = "julia_parentheses"

[julia_rainbow_paren_4]
inherit = "julia_rainbow_paren_1"

[julia_rainbow_paren_5]
inherit = "julia_rainbow_paren_2"

[julia_rainbow_paren_6]
inherit = "julia_rainbow_paren_3"

# Rainbow brackets
[julia_rainbow_bracket_1]
foreground = "#AE81FF"
inherit = "julia_parentheses"

[julia_rainbow_bracket_2]
foreground = "#E6DB74"
inherit = "julia_parentheses"

[julia_rainbow_bracket_3]
inherit = "julia_rainbow_bracket_1"
```

```
[julia_rainbow_bracket_4]
inherit = "julia_rainbow_bracket_2"

[julia_rainbow_bracket_5]
inherit = "julia_rainbow_bracket_1"

[julia_rainbow_bracket_6]
inherit = "julia_rainbow_bracket_2"

# Rainbow curlies
[julia_rainbow_curly_1]
foreground = "#F92672"
inherit = "julia_parentheses"

[julia_rainbow_curly_2]
foreground = "#A6E22E"
inherit = "julia_parentheses"

[julia_rainbow_curly_3]
inherit = "julia_rainbow_curly_1"

[julia_rainbow_curly_4]
inherit = "julia_rainbow_curly_2"

[julia_rainbow_curly_5]
inherit = "julia_rainbow_curly_1"

[julia_rainbow_curly_6]
inherit = "julia_rainbow_curly_2"
```

</details>

For a complete list of customizable faces, see the JuliaSyntaxHighlighting package documentation.

93.5 Customising the history searcher

The history searcher uses the following default faces, that can be customised:

```
[REPL.History.search]
separator.fg  = "blue"
prefix.fg = "magenta"
selected.fg = "blue"
unselected.fg = "grey"
hint = { fg = "magenta", slant = "italic", weight ="light" }
results.inherit = "shadow"
match = { weight = "bold", underline = true }
```

93.6 Customizing Colors

The colors used by Julia and the REPL can be customized, as well. To change the color of the Julia prompt you can add something like the following to your `~/.julia/config/startup.jl` file, which is to be placed inside your home directory:

```
function customize_colors(repl)
    repl.prompt_color = Base.text_colors[:cyan]
end

atreplinit(customize_colors)
```

The available color keys can be seen by typing `Base.text_colors` in the help mode of the REPL. In addition, the integers 0 to 255 can be used as color keys for terminals with 256 color support.

You can also change the colors for the help and shell prompts and input and answer text by setting the appropriate field of `repl` in the `customize_colors` function above (respectively, `help_color`, `shell_color`, `input_color`, and `answer_color`). For the latter two, be sure that the `envcolors` field is also set to false.

It is also possible to apply boldface formatting by using `Base.text_colors[:bold]` as a color. For instance, to print answers in boldface font, one can use the following as a `~/.julia/config/startup.jl`:

```
function customize_colors(repl)
    repl.envcolors = false
    repl.answer_color = Base.text_colors[:bold]
end

atreplinit(customize_colors)
```

You can also customize the color used to render warning and informational messages by setting the appropriate environment variables. For instance, to render error, warning, and informational messages respectively in magenta, yellow, and cyan you can add the following to your `~/.julia/config/startup.jl` file:

```
ENV["JULIA_ERROR_COLOR"] = :magenta
ENV["JULIA_WARN_COLOR"] = :yellow
ENV["JULIA_INFO_COLOR"] = :cyan
```

93.7 Changing the contextual module which is active at the REPL

When entering expressions at the REPL, they are by default evaluated in the `Main` module;

```
julia> @__MODULE__
Main
```

It is possible to change this contextual module via the function `REPL.activate(m)` where `m` is a `Module` or by typing the module in the REPL and pressing the keybinding Alt-m with the cursor on the module name (Esc-m on MacOS). Pressing the keybinding on an empty prompt toggles the context between the previously active non-`Main` module and `Main`. The active module is shown in the prompt (unless it is `Main`):

```
julia> using REPL

julia> REPL.activate(Base)

(Base) julia> @__MODULE__
```

```
Base

(Base) julia> using REPL # Need to load REPL into Base module to use it

(Base) julia> REPL.activate(Main)

julia>

julia> Core<Alt-m> # using the keybinding to change module

(Core) julia>

(Core) julia> <Alt-m> # going back to Main via keybinding

julia>

julia> <Alt-m> # going back to previously-active Core via keybinding

(Core) julia>
```

Functions that take an optional module argument often defaults to the REPL context module. As an example, calling `varinfo()` will show the variables of the current active module:

```
julia> module CustomMod
           export var, f
           var = 1
           f(x) = x^2
       end;

julia> REPL.activate(CustomMod)

(Main.CustomMod) julia> varinfo()
  name         size summary
  ———— ———————— ———————————————————————————
  CustomMod         Module
  f         0 bytes f (generic function with 1 method)
  var       8 bytes Int64
```

93.8 Numbered prompt

It is possible to get an interface which is similar to the IPython REPL and the Mathematica notebook with numbered input prompts and output prefixes. This is done by calling `REPL.numbered_prompt!()`. If you want to have this enabled on startup, add

```
atreplinit() do repl
    @eval import REPL
    if !isdefined(repl, :interface)
        repl.interface = REPL.setup_interface(repl)
    end
    REPL.numbered_prompt!(repl)
end
```

to your `startup.jl` file. In numbered prompt the variable `Out[n]` (where `n` is an integer) can be used to refer to earlier results:

```
In [1]: 5 + 3
Out[1]: 8

In [2]: Out[1] + 5
Out[2]: 13

In [3]: Out
Out[3]: Dict{Int64, Any} with 2 entries:
  2 => 13
  1 => 8
```

Note

Since all outputs from previous REPL evaluations are saved in the `Out` variable, one should be careful if they are returning many large in-memory objects like arrays, since they will be protected from garbage collection so long as a reference to them remains in `Out`. If you need to remove references to objects in `Out`, you can clear the entire history it stores with `empty!(Out)`, or clear an individual entry with `Out[n] = nothing`.

93.9 TerminalMenus

TerminalMenus is a submodule of the Julia REPL and enables small, low-profile interactive menus in the terminal.

Examples

```
import REPL
using REPL.TerminalMenus

options = ["apple", "orange", "grape", "strawberry",
           "blueberry", "peach", "lemon", "lime"]
```

RadioMenu

The RadioMenu allows the user to select one option from the list. The `request` function displays the interactive menu and returns the index of the selected choice. If a user presses 'q' or `ctrl-c`, `request` will return a `-1`.

```
# `pagesize` is the number of items to be displayed at a time.
#  The UI will scroll if the number of options is greater
#   than the `pagesize`
menu = RadioMenu(options, pagesize=4)

# `request` displays the menu and returns the index after the
#   user has selected a choice
choice = request("Choose your favorite fruit:", menu)
```

```
if choice != -1
    println("Your favorite fruit is ", options[choice], "!")
else
    println("Menu canceled.")
end
```

Output:

```
Choose your favorite fruit:
^  grape
   strawberry
 > blueberry
v  peach
Your favorite fruit is blueberry!
```

MultiSelectMenu

The MultiSelectMenu allows users to select many choices from a list.

```
# here we use the default `pagesize` 10
menu = MultiSelectMenu(options)

# `request` returns a `Set` of selected indices
# if the menu us canceled (ctrl-c or q), return an empty set
choices = request("Select the fruits you like:", menu)

if length(choices) > 0
    println("You like the following fruits:")
    for i in choices
        println("  - ", options[i])
    end
else
    println("Menu canceled.")
end
```

Output:

```
Select the fruits you like:
[press: Enter=toggle, a=all, n=none, d=done, q=abort]
   [ ] apple
 > [X] orange
   [X] grape
   [ ] strawberry
   [ ] blueberry
   [X] peach
   [ ] lemon
   [ ] lime
You like the following fruits:
  - orange
  - grape
  - peach
```

Customization / Configuration

ConfiguredMenu subtypes

Starting with Julia 1.6, the recommended way to configure menus is via the constructor. For instance, the default multiple-selection menu

```
julia> menu = MultiSelectMenu(options, pagesize=5);

julia> request(menu) # ASCII is used by default
[press: Enter=toggle, a=all, n=none, d=done, q=abort]
   [ ] apple
   [X] orange
   [ ] grape
 > [X] strawberry
v  [ ] blueberry
```

can instead be rendered with Unicode selection and navigation characters with

```
julia> menu = MultiSelectMenu(options, pagesize=5, charset=:unicode);

julia> request(menu)
[press: Enter=toggle, a=all, n=none, d=done, q=abort]
   ⬚ apple
   ✓ orange
   ⬚ grape
 → ✓ strawberry
↓  ⬚ blueberry
```

More fine-grained configuration is also possible:

```
julia> menu = MultiSelectMenu(options, pagesize=5, charset=:unicode, checked="YEP!",
↪  unchecked="NOPE", cursor='');

julia> request(menu)
julia> request(menu)
[press: Enter=toggle, a=all, n=none, d=done, q=abort]
   NOPE apple
   YEP! orange
   NOPE grape
  YEP! strawberry
↓  NOPE blueberry
```

Aside from the overall `charset` option, for `RadioMenu` the configurable options are:

- `cursor::Char='>'|'→'`: character to use for cursor
- `up_arrow::Char='^'|'↑'`: character to use for up arrow
- `down_arrow::Char='v'|'↓'`: character to use for down arrow
- `updown_arrow::Char='I'|'↕'`: character to use for up/down arrow in one-line page

- `scroll_wrap::Bool=false`: optionally wrap-around at the beginning/end of a menu
- `ctrl_c_interrupt::Bool=true`: If `false`, return empty on ^C, if `true` throw InterruptException() on ^C

`MultiSelectMenu` adds:

- `checked::String="[X]"|"✓"`: string to use for checked
- `unchecked::String="[ ]"|"⬚")`: string to use for unchecked

You can create new menu types of your own. Types that are derived from `TerminalMenus.ConfiguredMenu` configure the menu options at construction time.

Legacy interface

Prior to Julia 1.6, and still supported throughout Julia 1.x, one can also configure menus by calling `TerminalMenus.config()`.

93.10 References

REPL

`Base.atreplinit` – Function.

```
atreplinit(f)
```

Register a one-argument function to be called before the REPL interface is initialized in interactive sessions; this is useful to customize the interface. The argument of `f` is the REPL object. This function should be called from within the `.julia/config/startup.jl` initialization file.

source

TerminalMenus

Menus

`REPL.TerminalMenus.RadioMenu` – Type.

```
RadioMenu
```

A menu that allows a user to select a single option from a list.

Sample Output

```
julia> request(RadioMenu(options, pagesize=4))
Choose your favorite fruit:
^  grape
   strawberry
 > blueberry
v  peach
Your favorite fruit is blueberry!
```

REPL.TerminalMenus.MultiSelectMenu – Type.

```
MultiSelectMenu
```

A menu that allows a user to select a multiple options from a list.

Sample Output

```
julia> request(MultiSelectMenu(options))
Select the fruits you like:
[press: Enter=toggle, a=all, n=none, d=done, q=abort]
   [ ] apple
 > [X] orange
   [X] grape
   [ ] strawberry
   [ ] blueberry
   [X] peach
   [ ] lemon
   [ ] lime
You like the following fruits:
  - orange
  - grape
  - peach
```

Configuration

REPL.TerminalMenus.Config – Type.

```
Config(; scroll_wrap=false, ctrl_c_interrupt=true, charset=:ascii, cursor::Char,
↪  up_arrow::Char, down_arrow::Char)
```

Configure behavior for selection menus via keyword arguments:

- `scroll_wrap`, if `true`, causes the menu to wrap around when scrolling above the first or below the last entry
- `ctrl_c_interrupt`, if `true`, throws an `InterruptException` if the user hits Ctrl-C during menu selection. If `false`, `TerminalMenus.request` will return the default result from `TerminalMenus.selected`.
- `charset` affects the default values for `cursor`, `up_arrow`, and `down_arrow`, and can be `:ascii` or `:unicode`
- `cursor` is the character printed to indicate the option that will be chosen by hitting "Enter." Defaults are '>' or '→', depending on `charset`.
- `up_arrow` is the character printed when the display does not include the first entry. Defaults are '^' or '↑', depending on `charset`.
- `down_arrow` is the character printed when the display does not include the last entry. Defaults are 'v' or '↓', depending on `charset`.

Subtypes of `ConfiguredMenu` will print `cursor`, `up_arrow`, and `down_arrow` automatically as needed, your `writeline` method should not print them.

Julia 1.6

`Config` is available as of Julia 1.6. On older releases use the global `CONFIG`.

REPL.TerminalMenus.MultiSelectConfig – Type.

```
MultiSelectConfig(; charset=:ascii, checked::String, unchecked::String, kwargs...)
```

Configure behavior for a multiple-selection menu via keyword arguments:

- checked is the string to print when an option has been selected. Defaults are "[X]" or "✓", depending on charset.
- unchecked is the string to print when an option has not been selected. Defaults are "[ ]" or "⬚", depending on charset.

All other keyword arguments are as described for TerminalMenus.Config. checked and unchecked are not printed automatically, and should be printed by your writeline method.

Julia 1.6

MultiSelectConfig is available as of Julia 1.6. On older releases use the global CONFIG.

REPL.TerminalMenus.config – Function.

```
config( <see arguments> )
```

Keyword-only function to configure global menu parameters

Arguments

- charset::Symbol=:na: ui characters to use (:ascii or :unicode); overridden by other arguments
- cursor::Char='>'|'→': character to use for cursor
- up_arrow::Char='^'|'↑': character to use for up arrow
- down_arrow::Char='v'|'↓': character to use for down arrow
- checked::String="[X]"|"✓": string to use for checked
- unchecked::String="[ ]"|"⬚"): string to use for unchecked
- scroll::Symbol=:nowrap: If :wrap wrap cursor around top and bottom, if :nowrap do not wrap cursor
- supress_output::Bool=false: Ignored legacy argument, pass suppress_output as a keyword argument to request instead.
- ctrl_c_interrupt::Bool=true: If false, return empty on ^C, if true throw InterruptException() on ^C

Julia 1.6

As of Julia 1.6, config is deprecated. Use Config or MultiSelectConfig instead.

User interaction

REPL.TerminalMenus.request – Function.

```
request(m::AbstractMenu; cursor=1)
```

Display the menu and enter interactive mode. `cursor` indicates the item number used for the initial cursor position. `cursor` can be either an `Int` or a `RefValue{Int}`. The latter is useful for observation and control of the cursor position from the outside.

Returns `selected(m)`.

Julia 1.6

The `cursor` argument requires Julia 1.6 or later.

```
request([term,] msg::AbstractString, m::AbstractMenu)
```

Shorthand for `println(msg); request(m)`.

AbstractMenu extension interface

Any subtype of `AbstractMenu` must be mutable, and must contain the fields `pagesize::Int` and `pageoffset::Int`. Any subtype must also implement the following functions:

REPL.TerminalMenus.pick – Function.

```
pick(m::AbstractMenu, cursor::Int)
```

Defines what happens when a user presses the Enter key while the menu is open. If `true` is returned, `request()` will exit. `cursor` indexes the position of the selection.

REPL.TerminalMenus.cancel – Function.

```
cancel(m::AbstractMenu)
```

Define what happens when a user cancels ('q' or ctrl-c) a menu. `request()` will always exit after calling this function.

REPL.TerminalMenus.writeline – Function.

```
writeline(buf::IO, m::AbstractMenu, idx::Int, iscursor::Bool)
```

Write the option at index `idx` to `buf`. `iscursor`, if `true`, indicates that this item is at the current cursor position (the one that will be selected by hitting "Enter").

If `m` is a `ConfiguredMenu`, `TerminalMenus` will print the cursor indicator. Otherwise the callee is expected to handle such printing.

Julia 1.6

`writeline` requires Julia 1.6 or higher.

On older versions of Julia, this was `writeLine(buf::IO, m::AbstractMenu, idx, iscursor::Bool)` and `m` is assumed to be unconfigured. The selection and cursor indicators can be obtained from `TerminalMenus.CONFIG`.

This older function is supported on all Julia 1.x versions but will be dropped in Julia 2.0.

It must also implement either `options` or `numoptions`:

`REPL.TerminalMenus.options` – Function.

```
options(m::AbstractMenu)
```

Return a list of strings to be displayed as options in the current page.

Alternatively, implement `numoptions`, in which case `options` is not needed.

`REPL.TerminalMenus.numoptions` – Function.

```
numoptions(m::AbstractMenu)::Int
```

Return the number of options in menu `m`. Defaults to `length(options(m))`.

Julia 1.6

This function requires Julia 1.6 or later.

If the subtype does not have a field named `selected`, it must also implement

`REPL.TerminalMenus.selected` – Function.

```
selected(m::AbstractMenu)
```

Return information about the user-selected option. By default it returns `m.selected`.

The following are optional but can allow additional customization:

`REPL.TerminalMenus.header` – Function.

```
header(m::AbstractMenu)::String
```

Return a header string to be printed above the menu. Defaults to `""`.

`REPL.TerminalMenus.keypress` – Function.

```
keypress(m::AbstractMenu, i::UInt32)::Bool
```

Handle any non-standard keypress event. If `true` is returned, `TerminalMenus.request` will exit. Defaults to `false`.

Keybinding	Description
Program control	
^D	Exit (when buffer is empty)
^C	Interrupt or cancel
^L	Clear console screen
Return/Enter, ^J	New line, executing if it is complete
meta-Return/Enter	Insert new line without executing it
? or ;	Enter help or shell mode (when at start of a line)
^R, ^S	Interactive history search, described above
Cursor movement	
Right arrow, ^F	Move right one character
Left arrow, ^B	Move left one character
ctrl-Right, meta-F	Move right one word
ctrl-Left, meta-B	Move left one word
Home, ^A	Move to beginning of line
End, ^E	Move to end of line
Up arrow, ^P	Move up one line (or change to the previous history entry that matches the text before the cursor)
Down arrow, ^N	Move down one line (or change to the next history entry that matches the text before the cursor)
Shift-Arrow Key	Move cursor according to the direction of the Arrow key, while activating the region ("shift selection")
Page-up, meta-P	Change to the previous history entry
Page-down, meta-N	Change to the next history entry
meta-<	Change to the first history entry (of the current session if it is before the current position in history)
meta->	Change to the last history entry
^-Space	Set the "mark" in the editing region (and de-activate the region if it's active)
^-Space ^-Space	Set the "mark" in the editing region and make the region "active", i.e. highlighted
^G	De-activate the region (i.e. make it not highlighted)
^X^X	Exchange the current position with the mark
Editing	
Backspace, ^H	Delete the previous character, or the whole region when it's active
Delete, ^D	Forward delete one character (when buffer has text)
meta-Backspace	Delete the previous word
meta-d	Forward delete the next word
^W	Delete previous text up to the nearest whitespace
meta-w	Copy the current region in the kill ring
meta-W	"Kill" the current region, placing the text in the kill ring
^U	"Kill" to beginning of line, placing the text in the kill ring
^K	"Kill" to end of line, placing the text in the kill ring
^Y	"Yank" insert the text from the kill ring
meta-y	Replace a previously yanked text with an older entry from the kill ring
^T	Transpose the characters about the cursor
meta-Up arrow	Transpose current line with line above
meta-Down arrow	Transpose current line with line below
meta-u	Change the next word to uppercase
meta-c	Change the next word to titlecase
meta-l	Change the next word to lowercase
^/, ^_	Undo previous editing action
^Q	Write a number in REPL and press ^Q to open editor at corresponding stackframe or method
meta-Left	Indent the current line on the left

Chapter 94

Serialization

Provides serialization of Julia objects.

`Serialization.serialize` – Function.

```
serialize(stream::IO, value)
```

Write an arbitrary value to a stream in an opaque format, such that it can be read back by `deserialize`. The read-back value will be as identical as possible to the original, but note that `Ptr` values are serialized as all-zero bit patterns (`NULL`).

An 8-byte identifying header is written to the stream first. To avoid writing the header, construct a `Serializer` and use it as the first argument to `serialize` instead. See also `Serialization.writeheader`.

The data format can change in minor (1.x) Julia releases, but files written by prior 1.x versions will remain readable. The main exception to this is when the definition of a type in an external package changes. If that occurs, it may be necessary to specify an explicit compatible version of the affected package in your environment. Renaming functions, even private functions, inside packages can also put existing files out of sync. Anonymous functions require special care: because their names are automatically generated, minor code changes can cause them to be renamed. Serializing anonymous functions should be avoided in files intended for long-term storage.

In some cases, the word size (32- or 64-bit) of the reading and writing machines must match. In rarer cases the OS or architecture must also match, for example when using packages that contain platform-dependent code.

```
serialize(filename::AbstractString, value)
```

Open a file and serialize the given value to it.

Julia 1.1

This method is available as of Julia 1.1.

`Serialization.deserialize` – Function.

```
deserialize(stream)
```

Read a value written by `serialize`. `deserialize` assumes the binary data read from `stream` is correct and has been serialized by a compatible implementation of `serialize`. `deserialize` is designed for simplicity and performance, and so does not validate the data read. Malformed data can result in process termination. The caller must ensure the integrity and correctness of data read from `stream`.

```
deserialize(filename::AbstractString)
```

Open a file and deserialize its contents.

Julia 1.1

This method is available as of Julia 1.1.

`Serialization.writeheader` – Function.

```
Serialization.writeheader(s::AbstractSerializer)
```

Write an identifying header to the specified serializer. The header consists of 8 bytes as follows:

Offset	Description
0	tag byte (0x37)
1-2	signature bytes "JL"
3	protocol version
4	bits 0-1: endianness: 0 = little, 1 = big
4	bits 2-3: platform: 0 = 32-bit, 1 = 64-bit
5-7	reserved

Recommended File Extension

While the Serialization module does not mandate a specific file extension, the Julia community commonly uses the `.jls` extension for serialized Julia files.

Example:

```
open("model.jls", "w") do io
    serialize(io, my_model)
end
```

Chapter 95

SHA

95.1 SHA functions

Usage is very straightforward:

```
julia> using SHA

julia> bytes2hex(sha256("test"))
"9f86d081884c7d659a2feaa0c55ad015a3bf4f1b2b0b822cd15d6c15b0f00a08"
```

Each exported function (at the time of this writing, SHA-1, SHA-2 224, 256, 384 and 512, and SHA-3 224, 256, 384 and 512 functions are implemented) takes in either an `AbstractVector{UInt8}`, an `AbstractString` or an `IO` object. This makes it trivial to checksum a file:

```
shell> cat /tmp/test.txt
test
julia> using SHA

julia> open("/tmp/test.txt") do f
           sha2_256(f)
       end
32-element Vector{UInt8}:
 0x9f
 0x86
 0xd0
 0x81
 0x88
 0x4c
 0x7d
 0x65
    
 0x5d
 0x6c
 0x15
 0xb0
 0xf0
 0x0a
 0x08
```

All SHA functions

Due to the colloquial usage of sha256 to refer to sha2_256, convenience functions are provided, mapping shaxxx() function calls to sha2_xxx(). For SHA-3, no such colloquialisms exist and the user must use the full sha3_xxx() names.

shaxxx() takes AbstractString and array-like objects (NTuple and Vector) with elements of type UInt8.

SHA-1

SHA.sha1 – Function.

```
sha1(data)
```

Hash data using the sha1 algorithm and return the resulting digest. See also SHA1_CTX.

```
sha1(io::IO)
```

Hash data from io using sha1 algorithm.

SHA-2

SHA.sha224 – Function.

```
sha224(data)
```

Hash data using the sha224 algorithm and return the resulting digest. See also SHA2_224_CTX.

```
sha224(io::IO)
```

Hash data from io using sha224 algorithm.

SHA.sha256 – Function.

```
sha256(data)
```

Hash data using the sha256 algorithm and return the resulting digest. See also SHA2_256_CTX.

```
sha256(io::IO)
```

Hash data from io using sha256 algorithm.

SHA.sha384 – Function.

```
sha384(data)
```

Hash data using the sha384 algorithm and return the resulting digest. See also SHA2_384_CTX.

```
sha384(io::IO)
```

Hash data from io using sha384 algorithm.

SHA.sha512 – Function.

```
sha512(data)
```

Hash data using the sha512 algorithm and return the resulting digest. See also SHA2_512_CTX.

```
sha512(io::IO)
```

Hash data from io using sha512 algorithm.

SHA.sha2_224 – Function.

```
sha2_224(data)
```

Hash data using the sha2_224 algorithm and return the resulting digest. See also SHA2_224_CTX.

```
sha2_224(io::IO)
```

Hash data from io using sha2_224 algorithm.

SHA.sha2_256 – Function.

```
sha2_256(data)
```

Hash data using the sha2_256 algorithm and return the resulting digest. See also SHA2_256_CTX.

```
sha2_256(io::IO)
```

Hash data from io using sha2_256 algorithm.

SHA.sha2_384 – Function.

```
sha2_384(data)
```

Hash data using the sha2_384 algorithm and return the resulting digest. See also SHA2_384_CTX.

```
sha2_384(io::IO)
```

Hash data from io using sha2_384 algorithm.

SHA.sha2_512 – Function.

```
sha2_512(data)
```

Hash data using the sha2_512 algorithm and return the resulting digest. See also SHA2_512_CTX.

```
sha2_512(io::IO)
```

Hash data from io using sha2_512 algorithm.

SHA.sha2_512_224 – Function.

```
sha2_512_224(data)
```

Hash data using the sha2_512_224 algorithm and return the resulting digest. See also SHA2_512_224_CTX.

```
sha2_512_224(io::IO)
```

Hash data from io using sha2_512_224 algorithm.

SHA.sha2_512_256 – Function.

```
sha2_512_256(data)
```

Hash data using the sha2_512_256 algorithm and return the resulting digest. See also SHA2_512_256_CTX.

```
sha2_512_256(io::IO)
```

Hash data from io using sha2_512_256 algorithm.

SHA-3

SHA.sha3_224 – Function.

```
sha3_224(data)
```

Hash data using the sha3_224 algorithm and return the resulting digest. See also SHA3_224_CTX.

```
sha3_224(io::IO)
```

Hash data from io using sha3_224 algorithm.

SHA.sha3_256 – Function.

```
sha3_256(data)
```

Hash data using the sha3_256 algorithm and return the resulting digest. See also SHA3_256_CTX.

```
sha3_256(io::IO)
```

Hash data from io using sha3_256 algorithm.

SHA.sha3_384 – Function.

```
sha3_384(data)
```

Hash data using the sha3_384 algorithm and return the resulting digest. See also SHA3_384_CTX.

```
sha3_384(io::IO)
```

Hash data from io using sha3_384 algorithm.

SHA.sha3_512 – Function.

```
sha3_512(data)
```

Hash data using the sha3_512 algorithm and return the resulting digest. See also SHA3_512_CTX.

```
sha3_512(io::IO)
```

Hash data from io using sha3_512 algorithm.

95.2 Working with context

To create a hash from multiple items the `SHAX_XXX_CTX()` types can be used to create a stateful hash object that is updated with `update!` and finalized with `digest!`

```
julia> using SHA

julia> ctx = SHA2_256_CTX()
SHA2 256-bit hash state

julia> update!(ctx, b"some data")
0x0000000000000009

julia> update!(ctx, b"some more data")
0x0000000000000017

julia> digest!(ctx)
32-element Vector{UInt8}:
 0xbe
 0xcf
 0x23
 0xda
 0xaf
 0x02
 0xf7
 0xa3
 0x57
 0x92
    
 0x89
 0x4f
 0x59
 0xd8
 0xb3
 0xb4
 0x81
 0x8b
 0xc5
```

Note that, at the time of this writing, the SHA3 code is not optimized, and as such is roughly an order of magnitude slower than SHA2.

`SHA.update!` – Function.

```
update!(context, data[, datalen])
```

Update the SHA context with the bytes in data. See also `digest!` for finalizing the hash.

Examples

```
julia> ctx = SHA1_CTX()
SHA1 hash state

julia> update!(ctx, b"data to to be hashed")
```

SHA.digest! – Function.

```
digest!(context)
```

Finalize the SHA context and return the hash as array of bytes (Vector{Uint8}). Updating the context after calling digest! on it will error.

Examples

```
julia> ctx = SHA1_CTX()
SHA1 hash state

julia> update!(ctx, b"data to to be hashed")

julia> digest!(ctx)
20-element Vector{UInt8}:
 0x83
 0xe4
 
 0x89
 0xf5

julia> update!(ctx, b"more data")
ERROR: Cannot update CTX after `digest!` has been called on it
[...]
```

All SHA context types

SHA-1

SHA.SHA1_CTX – Type.

```
SHA1_CTX()
```

Construct an empty SHA1 context.

SHA-2

Convenience types are also provided, where SHAXXX_CTX is a type alias for SHA2_XXX_CTX.

SHA.SHA224_CTX – Type.

```
SHA2_224_CTX()
```

Construct an empty SHA2_224 context.

SHA.SHA256_CTX – Type.

```
SHA2_256_CTX()
```

Construct an empty SHA2_256 context.

SHA.SHA384_CTX – Type.

```
SHA2_384()
```

Construct an empty SHA2_384 context.

SHA.SHA512_CTX – Type.

```
SHA2_512_CTX()
```

Construct an empty SHA2_512 context.

SHA.SHA2_224_CTX – Type.

```
SHA2_224_CTX()
```

Construct an empty SHA2_224 context.

SHA.SHA2_256_CTX – Type.

```
SHA2_256_CTX()
```

Construct an empty SHA2_256 context.

SHA.SHA2_384_CTX – Type.

```
SHA2_384()
```

Construct an empty SHA2_384 context.

SHA.SHA2_512_CTX – Type.

```
SHA2_512_CTX()
```

Construct an empty SHA2_512 context.

SHA.SHA2_512_224_CTX – Type.

```
SHA2_512_224_CTX()
```

Construct an empty SHA2_512/224 context and set the initial hash value.

For the source of the initial value, refer to FIPS 180-4, 5.3.6.1 SHA-512/224

SHA.SHA2_512_256_CTX – Type.

```
SHA2_512_256_CTX()
```

Construct an empty SHA2_512/256 context and set the initial hash value.

For the source of the initial value, refer to FIPS 180-4, 5.3.6.2 SHA-512/256

SHA-3

SHA.SHA3_224_CTX – Type.

```
SHA3_224_CTX()
```

Construct an empty SHA3_224 context.

SHA.SHA3_256_CTX – Type.

```
SHA3_256_CTX()
```

Construct an empty SHA3_256 context.

SHA.SHA3_384_CTX – Type.

```
SHA3_384_CTX()
```

Construct an empty SHA3_384 context.

SHA.SHA3_512_CTX – Type.

```
SHA3_512_CTX()
```

Construct an empty SHA3_512 context.

95.3 HMAC functions

```
julia> using SHA

julia> key = collect(codeunits("key_string"))
10-element Vector{UInt8}:
 0x6b
 0x65
 0x79
 0x5f
 0x73
 0x74
 0x72
 0x69
 0x6e
 0x67

julia> bytes2hex(hmac_sha3_256(key, "test-message"))
"bc49a6f2aa29b27ee5ed1e944edd7f3d153e8a01535d98b5e24dac9a589a6248"
```

To create a hash from multiple items, the `HMAC_CTX()` types can be used to create a stateful hash object that is updated with `update!` and finalized with `digest!`.

```
julia> using SHA

julia> key = collect(codeunits("key_string"))
10-element Vector{UInt8}:
 0x6b
 0x65
 0x79
 0x5f
 0x73
 0x74
 0x72
 0x69
 0x6e
```

```
 0x67

julia> ctx = HMAC_CTX(SHA3_256_CTX(), key);

julia> update!(ctx, b"test-")
0x0000000000000000000000000000008d

julia> update!(ctx, b"message")
0x00000000000000000000000000000094

julia> bytes2hex(digest!(ctx))
"bc49a6f2aa29b27ee5ed1e944edd7f3d153e8a01535d98b5e24dac9a589a6248"
```

All HMAC functions

HMAC context type

SHA.HMAC_CTX – Type.

```
HMAC_CTX(ctx::CTX, key::Vector{UInt8}) where {CTX<:SHA_CTX}
```

Construct an empty HMAC_CTX context.

SHA-1

SHA.hmac_sha1 – Function.

```
hmac_sha1(key, data)
```

Hash data using the sha1 algorithm using the passed key. See also HMAC_CTX.

```
hmac_sha1(key, io::IO)
```

Hash data from io with the passed key using sha1 algorithm.

SHA-2

SHA.hmac_sha224 – Function.

```
hmac_sha224(key, data)
```

Hash data using the sha224 algorithm using the passed key. See also HMAC_CTX.

```
hmac_sha224(key, io::IO)
```

Hash data from io with the passed key using sha224 algorithm.

SHA.hmac_sha256 – Function.

```
hmac_sha256(key, data)
```

Hash data using the sha256 algorithm using the passed key. See also HMAC_CTX.

```
hmac_sha256(key, io::IO)
```

Hash data from io with the passed key using sha256 algorithm.

SHA.hmac_sha384 – Function.

```
hmac_sha384(key, data)
```

Hash data using the sha384 algorithm using the passed key. See also HMAC_CTX.

```
hmac_sha384(key, io::IO)
```

Hash data from io with the passed key using sha384 algorithm.

SHA.hmac_sha512 – Function.

```
hmac_sha512(key, data)
```

Hash data using the sha512 algorithm using the passed key. See also HMAC_CTX.

```
hmac_sha512(key, io::IO)
```

Hash data from io with the passed key using sha512 algorithm.

SHA.hmac_sha2_224 – Function.

```
hmac_sha2_224(key, data)
```

Hash data using the sha2_224 algorithm using the passed key. See also HMAC_CTX.

```
hmac_sha2_224(key, io::IO)
```

Hash data from io with the passed key using sha2_224 algorithm.

SHA.hmac_sha2_256 – Function.

```
hmac_sha2_256(key, data)
```

Hash data using the sha2_256 algorithm using the passed key. See also HMAC_CTX.

```
hmac_sha2_256(key, io::IO)
```

Hash data from io with the passed key using sha2_256 algorithm.

SHA.hmac_sha2_384 – Function.

```
hmac_sha2_384(key, data)
```

Hash data using the sha2_384 algorithm using the passed key. See also HMAC_CTX.

```
hmac_sha2_384(key, io::IO)
```

Hash data from io with the passed key using sha2_384 algorithm.

SHA.hmac_sha2_512 – Function.

```
hmac_sha2_512(key, data)
```

Hash data using the sha2_512 algorithm using the passed key. See also HMAC_CTX.

```
hmac_sha2_512(key, io::IO)
```

Hash data from io with the passed key using sha2_512 algorithm.

SHA-3

SHA.hmac_sha3_224 – Function.

```
hmac_sha3_224(key, data)
```

Hash data using the sha3_224 algorithm using the passed key. See also HMAC_CTX.

```
hmac_sha3_224(key, io::IO)
```

Hash data from io with the passed key using sha3_224 algorithm.

SHA.hmac_sha3_256 – Function.

```
hmac_sha3_256(key, data)
```

Hash data using the sha3_256 algorithm using the passed key. See also HMAC_CTX.

```
hmac_sha3_256(key, io::IO)
```

Hash data from io with the passed key using sha3_256 algorithm.

SHA.hmac_sha3_384 – Function.

```
hmac_sha3_384(key, data)
```

Hash data using the sha3_384 algorithm using the passed key. See also HMAC_CTX.

```
hmac_sha3_384(key, io::IO)
```

Hash data from io with the passed key using sha3_384 algorithm.

SHA.hmac_sha3_512 – Function.

```
hmac_sha3_512(key, data)
```

Hash data using the sha3_512 algorithm using the passed key. See also HMAC_CTX.

```
hmac_sha3_512(key, io::IO)
```

Hash data from io with the passed key using sha3_512 algorithm.

Chapter 96

Shared Arrays

SharedArray represents an array, which is shared across multiple processes, on a single machine.

SharedArrays.SharedArray – Type.

```
SharedArray{T}(dims::NTuple; init=false, pids=Int[])
SharedArray{T,N}(...)
```

Construct a SharedArray of a bits type T and size dims across the processes specified by pids - all of which have to be on the same host. If N is specified by calling SharedArray{T,N}(dims), then N must match the length of dims.

If pids is left unspecified, the shared array will be mapped across all processes on the current host, including the master. But, localindices and indexpids will only refer to worker processes. This facilitates work distribution code to use workers for actual computation with the master process acting as a driver.

If an init function of the type initfn(S::SharedArray) is specified, it is called on all the participating workers.

The shared array is valid as long as a reference to the SharedArray object exists on the node which created the mapping.

```
SharedArray{T}(filename::AbstractString, dims::NTuple, [offset=0]; mode=nothing, init=false,
↪  pids=Int[])
SharedArray{T,N}(...)
```

Construct a SharedArray backed by the file filename, with element type T (must be a bits type) and size dims, across the processes specified by pids - all of which have to be on the same host. This file is mmapped into the host memory, with the following consequences:

- The array data must be represented in binary format (e.g., an ASCII format like CSV cannot be supported)
- Any changes you make to the array values (e.g., A[3] = 0) will also change the values on disk

If pids is left unspecified, the shared array will be mapped across all processes on the current host, including the master. But, localindices and indexpids will only refer to worker processes. This facilitates work distribution code to use workers for actual computation with the master process acting as a driver.

mode must be one of "r", "r+", "w+", or "a+", and defaults to "r+" if the file specified by filename already exists, or "w+" if not. If an init function of the type initfn(S::SharedArray) is specified, it is called on all the participating workers. You cannot specify an init function if the file is not writable.

offset allows you to skip the specified number of bytes at the beginning of the file.

SharedArrays.SharedVector – Type.

```
SharedVector
```

A one-dimensional SharedArray.

SharedArrays.SharedMatrix – Type.

```
SharedMatrix
```

A two-dimensional SharedArray.

Distributed.procs – Method.

```
procs(S::SharedArray)
```

Get the vector of processes mapping the shared array.

SharedArrays.sdata – Function.

```
sdata(S::SharedArray)
```

Return the actual Array object backing S.

SharedArrays.indexpids – Function.

```
indexpids(S::SharedArray)
```

Return the current worker's index in the list of workers mapping the SharedArray (i.e. in the same list returned by procs(S)), or 0 if the SharedArray is not mapped locally.

SharedArrays.localindices – Function.

```
localindices(S::SharedArray)
```

Return a range describing the "default" indices to be handled by the current process. This range should be interpreted in the sense of linear indexing, i.e., as a sub-range of 1:length(S). In multi-process contexts, returns an empty range in the parent process (or any process for which indexpids returns 0).

It's worth emphasizing that localindices exists purely as a convenience, and you can partition work on the array among workers any way you wish. For a SharedArray, all indices should be equally fast for each worker process.

Chapter 97

Sockets

`Sockets.Sockets` – Module.

Support for sockets. Provides `IPAddr` and subtypes, `TCPSocket`, and `UDPSocket`.

`Sockets.connect` – Method.

```
connect([host], port::Integer) -> TCPSocket
```

Connect to the host `host` on port `port`.

`Sockets.connect` – Method.

```
connect(path::AbstractString) -> PipeEndpoint
```

Connect to the named pipe / UNIX domain socket at `path`.

Note

Path length on Unix is limited to somewhere between 92 and 108 bytes (cf. `man unix`).

`Sockets.listen` – Method.

```
listen([addr, ]port::Integer; backlog::Integer=BACKLOG_DEFAULT) -> TCPServer
```

Listen on port on the address specified by `addr`. By default this listens on `localhost` only. To listen on all interfaces pass `IPv4(0)` or `IPv6(0)` as appropriate. `backlog` determines how many connections can be pending (not having called `accept`) before the server will begin to reject them. The default value of `backlog` is 511.

`Sockets.listen` – Method.

```
listen(path::AbstractString) -> PipeServer
```

Create and listen on a named pipe / UNIX domain socket.

Note

Path length on Unix is limited to somewhere between 92 and 108 bytes (cf. `man unix`).

Sockets.getaddrinfo – Function.

```
getaddrinfo(host::AbstractString, IPAddr) -> IPAddr
```

Gets the first IP address of the host of the specified IPAddr type. Uses the operating system's underlying getaddrinfo implementation, which may do a DNS lookup.

Examples

```
julia> getaddrinfo("localhost", IPv6)
ip"::1"

julia> getaddrinfo("localhost", IPv4)
ip"127.0.0.1"
```

```
getaddrinfo(host::AbstractString) -> IPAddr
```

Gets the first available IP address of host, which may be either an IPv4 or IPv6 address. Uses the operating system's underlying getaddrinfo implementation, which may do a DNS lookup.

Sockets.getipaddr – Function.

```
getipaddr() -> IPAddr
```

Get an IP address of the local machine, preferring IPv4 over IPv6. Throws if no addresses are available.

```
getipaddr(addr_type::Type{T}) where T<:IPAddr -> T
```

Get an IP address of the local machine of the specified type. Throws if no addresses of the specified type are available.

This function is a backwards-compatibility wrapper around getipaddrs. New applications should use getipaddrs instead.

Examples

```
julia> getipaddr()
ip"192.168.1.28"

julia> getipaddr(IPv6)
ip"fe80::9731:35af:e1c5:6e49"
```

See also getipaddrs.

Sockets.getipaddrs – Function.

```
getipaddrs(addr_type::Type{T}=IPAddr; loopback::Bool=false) where T<:IPAddr -> Vector{T}
```

Get the IP addresses of the local machine.

Setting the optional addr_type parameter to IPv4 or IPv6 causes only addresses of that type to be returned.

The loopback keyword argument dictates whether loopback addresses (e.g. ip"127.0.0.1", ip"::1") are included.

Julia 1.2

This function is available as of Julia 1.2.

Examples

```
julia> getipaddrs()
5-element Vector{IPAddr}:
 ip"198.51.100.17"
 ip"203.0.113.2"
 ip"2001:db8:8:4:445e:5fff:fe5d:5500"
 ip"2001:db8:8:4:c164:402e:7e3c:3668"
 ip"fe80::445e:5fff:fe5d:5500"

julia> getipaddrs(IPv6)
3-element Vector{IPv6}:
 ip"2001:db8:8:4:445e:5fff:fe5d:5500"
 ip"2001:db8:8:4:c164:402e:7e3c:3668"
 ip"fe80::445e:5fff:fe5d:5500"
```

See also `islinklocaladdr`.

`Sockets.islinklocaladdr` – Function.

```
islinklocaladdr(addr::IPAddr)
```

Tests if an IP address is a link-local address. Link-local addresses are not guaranteed to be unique beyond their network segment, therefore routers do not forward them. Link-local addresses are from the address blocks `169.254.0.0/16` or `fe80::/10`.

Examples

```
filter(!islinklocaladdr, getipaddrs())
```

`Sockets.getalladdrinfo` – Function.

```
getalladdrinfo(host::AbstractString) -> Vector{IPAddr}
```

Gets all of the IP addresses of the `host`. Uses the operating system's underlying `getaddrinfo` implementation, which may do a DNS lookup.

Examples

```
julia> getalladdrinfo("google.com")
2-element Vector{IPAddr}:
 ip"172.217.6.174"
 ip"2607:f8b0:4000:804::200e"
```

`Sockets.DNSError` – Type.

```
DNSError
```

The type of exception thrown when an error occurs in DNS lookup. The `host` field indicates the host URL string. The code field indicates the error code based on libuv.

`Sockets.getnameinfo` – Function.

```
getnameinfo(host::IPAddr) -> String
```

Performs a reverse-lookup for IP address to return a hostname and service using the operating system's underlying `getnameinfo` implementation.

Examples

```
julia> getnameinfo(IPv4("8.8.8.8"))
"google-public-dns-a.google.com"
```

`Sockets.getsockname` – Function.

```
getsockname(sock::Union{TCPServer, TCPSocket}) -> (IPAddr, UInt16)
```

Get the IP address and port that the given socket is bound to.

`Sockets.getpeername` – Function.

```
getpeername(sock::TCPSocket) -> (IPAddr, UInt16)
```

Get the IP address and port of the remote endpoint that the given socket is connected to. Valid only for connected TCP sockets.

`Sockets.IPAddr` – Type.

```
IPAddr
```

Abstract supertype for IP addresses. `IPv4` and `IPv6` are subtypes of this.

`Sockets.IPv4` – Type.

```
IPv4(host::Integer) -> IPv4
```

Return an IPv4 object from IP address `host` formatted as an `Integer`.

Examples

```
julia> IPv4(3223256218)
ip"192.30.252.154"
```

```
IPv4(str::AbstractString) -> IPv4
```

Parse an IPv4 address string into an `IPv4` object.

Examples

```
julia> IPv4("127.0.0.1")
ip"127.0.0.1"
```

`Sockets.IPv6` – Type.

```
IPv6(host::Integer) -> IPv6
```

Return an IPv6 object from IP address `host` formatted as an `Integer`.

Examples

```
julia> IPv6(3223256218)
ip"::c01e:fc9a"
```

```
IPv6(str::AbstractString) -> IPv6
```

Parse an IPv6 address string into an `IPv6` object.

Examples

```
julia> IPv6("::1")
ip"::1"
```

`Sockets.@ip_str` – Macro.

```
@ip_str str -> IPAddr
```

Parse `str` as an IP address.

Examples

```
julia> ip"127.0.0.1"
ip"127.0.0.1"

julia> @ip_str "2001:db8:0:0:0:0:2:1"
ip"2001:db8::2:1"
```

`Sockets.TCPSocket` – Type.

```
TCPSocket(; delay=true)
```

Open a TCP socket using libuv. If `delay` is true, libuv delays creation of the socket's file descriptor till the first `bind` call. `TCPSocket` has various fields to denote the state of the socket as well as its send/receive buffers.

`Sockets.UDPSocket` – Type.

```
UDPSocket()
```

Open a UDP socket using libuv. `UDPSocket` has various fields to denote the state of the socket.

`Sockets.accept` – Function.

```
accept(server[, client])
```

Accepts a connection on the given server and returns a connection to the client. An uninitialized client stream may be provided, in which case it will be used instead of creating a new stream.

`Sockets.listenany` – Function.

```
listenany([host::IPAddr,] port_hint; backlog::Integer=BACKLOG_DEFAULT) -> (UInt16, TCPServer)
```

Create a TCPServer on any port, using hint as a starting point. Returns a tuple of the actual port that the server was created on and the server itself. The backlog argument defines the maximum length to which the queue of pending connections for sockfd may grow.

Base.bind – Function.

```
bind(chnl::Channel, task::Task)
```

Associate the lifetime of chnl with a task. Channel chnl is automatically closed when the task terminates. Any uncaught exception in the task is propagated to all waiters on chnl.

The chnl object can be explicitly closed independent of task termination. Terminating tasks have no effect on already closed Channel objects.

When a channel is bound to multiple tasks, the first task to terminate will close the channel. When multiple channels are bound to the same task, termination of the task will close all of the bound channels.

Examples

```
julia> c = Channel(0);

julia> task = @async foreach(i->put!(c, i), 1:4);

julia> bind(c,task);

julia> for i in c
           @show i
       end;
i = 1
i = 2
i = 3
i = 4

julia> isopen(c)
false
```

```
julia> c = Channel(0);

julia> task = @async (put!(c, 1); error("foo"));

julia> bind(c, task);

julia> take!(c)
1

julia> put!(c, 1);
ERROR: TaskFailedException
Stacktrace:
[...]
    nested task error: foo
[...]
```

source

```
bind(socket::Union{TCPServer, UDPSocket, TCPSocket}, host::IPAddr, port::Integer;
↪  ipv6only=false, reuseaddr=false, kws...)
```

Bind `socket` to the given `host:port`. Note that `0.0.0.0` will listen on all devices.

- The `ipv6only` parameter disables dual stack mode. If `ipv6only=true`, only an IPv6 stack is created.
- If `reuseaddr=true`, multiple threads or processes can bind to the same address without error if they all set `reuseaddr=true`, but only the last to bind will receive any traffic.

`Sockets.send` – Function.

```
send(socket::UDPSocket, host::IPAddr, port::Integer, msg)
```

Send `msg` over `socket` to `host:port`.

`Sockets.recv` – Function.

```
recv(socket::UDPSocket)
```

Read a UDP packet from the specified socket, and return the bytes received. This call blocks.

`Sockets.recvfrom` – Function.

```
recvfrom(socket::UDPSocket) -> (host_port, data)
```

Read a UDP packet from the specified socket, returning a tuple of `(host_port, data)`, where `host_port` will be an InetAddr{IPv4} or InetAddr{IPv6}, as appropriate.

Julia 1.3

Prior to Julia version 1.3, the first returned value was an address (`IPAddr`). In version 1.3 it was changed to an `InetAddr`.

`Sockets.setopt` – Function.

```
setopt(sock::UDPSocket; multicast_loop=nothing, multicast_ttl=nothing,
↪  enable_broadcast=nothing, ttl=nothing)
```

Set UDP socket options.

- `multicast_loop`: loopback for multicast packets (default: `true`).
- `multicast_ttl`: TTL for multicast packets (default: `nothing`).
- `enable_broadcast`: flag must be set to `true` if socket will be used for broadcast messages, or else the UDP system will return an access error (default: `false`).
- `ttl`: Time-to-live of packets sent on the socket (default: `nothing`).

`Sockets.nagle` – Function.

```
nagle(socket::Union{TCPServer, TCPSocket}, enable::Bool)
```

Nagle's algorithm batches multiple small TCP packets into larger ones. This can improve throughput but worsen latency. Nagle's algorithm is enabled by default. This function sets whether Nagle's algorithm is active on a given TCP server or socket. The opposite option is called `TCP_NODELAY` in other languages.

Julia 1.3

This function requires Julia 1.3 or later.

`Sockets.quickack` – Function.

```
quickack(socket::Union{TCPServer, TCPSocket}, enable::Bool)
```

On Linux systems, the TCP_QUICKACK is disabled or enabled on `socket`.

Chapter 98

Sparse Arrays

Julia has support for sparse vectors and sparse matrices in the `SparseArrays` stdlib module. Sparse arrays are arrays that contain enough zeros that storing them in a special data structure leads to savings in space and execution time, compared to dense arrays.

External packages which implement different sparse storage types, multidimensional sparse arrays, and more can be found in Noteworthy External Sparse Packages

98.1 Compressed Sparse Column (CSC) Sparse Matrix Storage

In Julia, sparse matrices are stored in the Compressed Sparse Column (CSC) format. Julia sparse matrices have the type `SparseMatrixCSC{Tv,Ti}`, where `Tv` is the type of the stored values, and `Ti` is the integer type for storing column pointers and row indices. The internal representation of `SparseMatrixCSC` is as follows:

```
struct SparseMatrixCSC{Tv,Ti<:Integer} <: AbstractSparseMatrixCSC{Tv,Ti}
    m::Int                  # Number of rows
    n::Int                  # Number of columns
    colptr::Vector{Ti}      # Column j is in colptr[j]:(colptr[j+1]-1)
    rowval::Vector{Ti}      # Row indices of stored values
    nzval::Vector{Tv}       # Stored values, typically nonzeros
end
```

The compressed sparse column storage makes it easy and quick to access the elements in the column of a sparse matrix, whereas accessing the sparse matrix by rows is considerably slower. Operations such as insertion of previously unstored entries one at a time in the CSC structure tend to be slow. This is because all elements of the sparse matrix that are beyond the point of insertion have to be moved one place over.

All operations on sparse matrices are carefully implemented to exploit the CSC data structure for performance, and to avoid expensive operations.

If you have data in CSC format from a different application or library, and wish to import it in Julia, make sure that you use 1-based indexing. The row indices in every column need to be sorted, and if they are not, the matrix will display incorrectly. If your `SparseMatrixCSC` object contains unsorted row indices, one quick way to sort them is by doing a double transpose. Since the transpose operation is lazy, make a copy to materialize each transpose.

In some applications, it is convenient to store explicit zero values in a SparseMatrixCSC. These *are* accepted by functions in Base (but there is no guarantee that they will be preserved in mutating operations). Such explicitly stored zeros are treated as structural nonzeros by many routines. The nnz function returns the number of elements explicitly stored in the sparse data structure, including non-structural zeros. In order to count the exact number of numerical nonzeros, use count(!iszero, x), which inspects every stored element of a sparse matrix. dropzeros, and the in-place dropzeros!, can be used to remove stored zeros from the sparse matrix.

```
julia> A = sparse([1, 1, 2, 3], [1, 3, 2, 3], [0, 1, 2, 0])
3×3 SparseMatrixCSC{Int64, Int64} with 4 stored entries:
 0  ·  1
 ·  2  ·
 ·  ·  0

julia> dropzeros(A)
3×3 SparseMatrixCSC{Int64, Int64} with 2 stored entries:
 ·  ·  1
 ·  2  ·
 ·  ·  ·
```

98.2 Sparse Vector Storage

Sparse vectors are stored in a close analog to compressed sparse column format for sparse matrices. In Julia, sparse vectors have the type SparseVector{Tv,Ti} where Tv is the type of the stored values and Ti the integer type for the indices. The internal representation is as follows:

```
struct SparseVector{Tv,Ti<:Integer} <: AbstractSparseVector{Tv,Ti}
    n::Int              # Length of the sparse vector
    nzind::Vector{Ti}   # Indices of stored values
    nzval::Vector{Tv}   # Stored values, typically nonzeros
end
```

Like SparseMatrixCSC, the SparseVector type can also contain explicitly stored zeros. (See Sparse Matrix Storage.).

98.3 Sparse Vector and Matrix Constructors

The simplest way to create a sparse array is to use a function equivalent to the zeros function that Julia provides for working with dense arrays. To produce a sparse array instead, you can use the same name with an sp prefix:

```
julia> spzeros(3)
3-element SparseVector{Float64, Int64} with 0 stored entries
```

The sparse function is often a handy way to construct sparse arrays. For example, to construct a sparse matrix we can input a vector I of row indices, a vector J of column indices, and a vector V of stored values (this is also known as the COO (coordinate) format). sparse(I,J,V) then constructs a sparse matrix such that S[I[k], J[k]] = V[k]. The equivalent sparse vector constructor is sparsevec, which takes the (row) index vector I and the vector V with the stored values and constructs a sparse vector R such that R[I[k]] = V[k].

```
julia> I = [1, 4, 3, 5]; J = [4, 7, 18, 9]; V = [1, 2, -5, 3];

julia> S = sparse(I,J,V)
5×18 SparseMatrixCSC{Int64, Int64} with 4 stored entries:
⎡⠀⠀⠀⠀⠀⠀⠀⠀⠀⠀⠀⠀⠀⠀⎤
⎣⠀⠀⠀⠀⠀⠀⠀⠀⠀⠀⠀⠀⠀⠀⎦

julia> R = sparsevec(I,V)
5-element SparseVector{Int64, Int64} with 4 stored entries:
  [1]  =  1
  [3]  =  -5
  [4]  =  2
  [5]  =  3
```

The inverse of the `sparse` and `sparsevec` functions is `findnz`, which retrieves the inputs used to create the sparse array (including stored entries equal to zero). `findall(!iszero, x)` returns the Cartesian indices of non-zero entries in x (not including stored entries equal to zero).

```
julia> findnz(S)
([1, 4, 5, 3], [4, 7, 9, 18], [1, 2, 3, -5])

julia> findall(!iszero, S)
4-element Vector{CartesianIndex{2}}:
 CartesianIndex(1, 4)
 CartesianIndex(4, 7)
 CartesianIndex(5, 9)
 CartesianIndex(3, 18)

julia> findnz(R)
([1, 3, 4, 5], [1, -5, 2, 3])

julia> findall(!iszero, R)
4-element Vector{Int64}:
 1
 3
 4
 5
```

Another way to create a sparse array is to convert a dense array into a sparse array using the `sparse` function:

```
julia> sparse(Matrix(1.0I, 5, 5))
5×5 SparseMatrixCSC{Float64, Int64} with 5 stored entries:
 1.0   ⋅    ⋅    ⋅    ⋅
  ⋅   1.0   ⋅    ⋅    ⋅
  ⋅    ⋅   1.0   ⋅    ⋅
  ⋅    ⋅    ⋅   1.0   ⋅
  ⋅    ⋅    ⋅    ⋅   1.0

julia> sparse([1.0, 0.0, 1.0])
3-element SparseVector{Float64, Int64} with 2 stored entries:
```

```
  [1]  =  1.0
  [3]  =  1.0
```

You can go in the other direction using the `Array` constructor. The `issparse` function can be used to query if a matrix is sparse.

```
julia> issparse(spzeros(5))
true
```

98.4 Sparse matrix operations

Arithmetic operations on sparse matrices also work as they do on dense matrices. Indexing of, assignment into, and concatenation of sparse matrices work in the same way as dense matrices. Indexing operations, especially assignment, are expensive, when carried out one element at a time. In many cases it may be better to convert the sparse matrix into `(I,J,V)` format using `findnz`, manipulate the values or the structure in the dense vectors `(I,J,V)`, and then reconstruct the sparse matrix.

98.5 Correspondence of dense and sparse methods

The following table gives a correspondence between built-in methods on sparse matrices and their corresponding methods on dense matrix types. In general, methods that generate sparse matrices differ from their dense counterparts in that the resulting matrix follows the same sparsity pattern as a given sparse matrix `S`, or that the resulting sparse matrix has density `d`, i.e. each matrix element has a probability `d` of being non-zero.

Details can be found in the Sparse Vectors and Matrices section of the standard library reference.

Sparse	Dense	Description
`spzeros(m,n)`	`zeros(m,n)`	Creates a *m*-by-*n* matrix of zeros. (`spzeros(m,n)` is empty.)
`sparse(I,n,n)`	`Matrix(I,n,n)`	Creates a *n*-by-*n* identity matrix.
`sparse(A)`	`Array(S)`	Interconverts between dense and sparse formats.
`sprand(m,n,d)`	`rand(m,n)`	Creates a *m*-by-*n* random matrix (of density *d*) with iid non-zero elements distributed uniformly on the half-open interval $[0, 1)$.
`sprandn(m,n,d)`	`randn(m,n)`	Creates a *m*-by-*n* random matrix (of density *d*) with iid non-zero elements distributed according to the standard normal (Gaussian) distribution.
`sprandn(rng,m,n,d)`	`randn(rng,m,n)`	Creates a *m*-by-*n* random matrix (of density *d*) with iid non-zero elements generated with the `rng` random number generator

Chapter 99

SparseArrays API

SparseArrays.AbstractSparseArray – Type.

```
AbstractSparseArray{Tv,Ti,N}
```

Supertype for N-dimensional sparse arrays (or array-like types) with elements of type Tv and index type Ti. SparseMatrixCSC, SparseVector and SuiteSparse.CHOLMOD.Sparse are subtypes of this.

SparseArrays.AbstractSparseVector – Type.

```
AbstractSparseVector{Tv,Ti}
```

Supertype for one-dimensional sparse arrays (or array-like types) with elements of type Tv and index type Ti. Alias for AbstractSparseArray{Tv,Ti,1}.

SparseArrays.AbstractSparseMatrix – Type.

```
AbstractSparseMatrix{Tv,Ti}
```

Supertype for two-dimensional sparse arrays (or array-like types) with elements of type Tv and index type Ti. Alias for AbstractSparseArray{Tv,Ti,2}.

SparseArrays.SparseVector – Type.

```
SparseVector{Tv,Ti<:Integer} <: AbstractSparseVector{Tv,Ti}
```

Vector type for storing sparse vectors. Can be created by passing the length of the vector, a *sorted* vector of non-zero indices, and a vector of non-zero values.

For instance, the vector [5, 6, 0, 7] can be represented as

```
SparseVector(4, [1, 2, 4], [5, 6, 7])
```

This indicates that the element at index 1 is 5, at index 2 is 6, at index 3 is zero(Int), and at index 4 is 7.

It may be more convenient to create sparse vectors directly from dense vectors using sparse as

```
sparse([5, 6, 0, 7])
```

yields the same sparse vector.

SparseArrays.SparseMatrixCSC – Type.

```
SparseMatrixCSC{Tv,Ti<:Integer} <: AbstractSparseMatrixCSC{Tv,Ti}
```

Matrix type for storing sparse matrices in the Compressed Sparse Column format. The standard way of constructing SparseMatrixCSC is through the `sparse` function. See also `spzeros`, `spdiagm` and `sprand`.

SparseArrays.sparse – Function.

```
sparse(A::Union{AbstractVector, AbstractMatrix})
```

Convert a vector or matrix `A` into a sparse array. Numerical zeros in `A` are turned into structural zeros.

Examples

```
julia> A = Matrix(1.0I, 3, 3)
3×3 Matrix{Float64}:
 1.0  0.0  0.0
 0.0  1.0  0.0
 0.0  0.0  1.0

julia> sparse(A)
3×3 SparseMatrixCSC{Float64, Int64} with 3 stored entries:
 1.0   ⋅    ⋅
  ⋅   1.0   ⋅
  ⋅    ⋅   1.0

julia> [1.0, 0.0, 1.0]
3-element Vector{Float64}:
 1.0
 0.0
 1.0

julia> sparse([1.0, 0.0, 1.0])
3-element SparseVector{Float64, Int64} with 2 stored entries:
  [1]  =  1.0
  [3]  =  1.0
```

```
sparse(I, J, V,[ m, n, combine])
```

Create a sparse matrix `S` of dimensions `m x n` such that `S[I[k], J[k]] = V[k]`. The `combine` function is used to combine duplicates. If `m` and `n` are not specified, they are set to `maximum(I)` and `maximum(J)` respectively. If the `combine` function is not supplied, `combine` defaults to `+` unless the elements of `V` are Booleans in which case `combine` defaults to `|`. All elements of `I` must satisfy `1 <= I[k] <= m`, and all elements of `J` must satisfy `1 <= J[k] <= n`. Numerical zeros in `(I, J, V)` are retained as structural nonzeros; to drop numerical zeros, use `dropzeros!`.

For additional documentation and an expert driver, see `SparseArrays.sparse!`.

Examples

```
julia> Is = [1; 2; 3];

julia> Js = [1; 2; 3];

julia> Vs = [1; 2; 3];

julia> sparse(Is, Js, Vs)
3×3 SparseMatrixCSC{Int64, Int64} with 3 stored entries:
 1  ·  ·
 ·  2  ·
 ·  ·  3
```

SparseArrays.sparse! – Function.

```
sparse!(I::AbstractVector{Ti}, J::AbstractVector{Ti}, V::AbstractVector{Tv},
        m::Integer, n::Integer, combine, klasttouch::Vector{Ti},
        csrrowptr::Vector{Ti}, csrcolval::Vector{Ti}, csrnzval::Vector{Tv},
        [csccolptr::Vector{Ti}], [cscrowval::Vector{Ti}, cscnzval::Vector{Tv}] ) where
        ↪  {Tv,Ti<:Integer}
```

Parent of and expert driver for sparse; see sparse for basic usage. This method allows the user to provide preallocated storage for sparse's intermediate objects and result as described below. This capability enables more efficient successive construction of SparseMatrixCSCs from coordinate representations, and also enables extraction of an unsorted-column representation of the result's transpose at no additional cost.

This method consists of three major steps: (1) Counting-sort the provided coordinate representation into an unsorted-row CSR form including repeated entries. (2) Sweep through the CSR form, simultaneously calculating the desired CSC form's column-pointer array, detecting repeated entries, and repacking the CSR form with repeated entries combined; this stage yields an unsorted-row CSR form with no repeated entries. (3) Counting-sort the preceding CSR form into a fully-sorted CSC form with no repeated entries.

Input arrays csrrowptr, csrcolval, and csrnzval constitute storage for the intermediate CSR forms and require length(csrrowptr) >= m + 1, length(csrcolval) >= length(I), and length(csrnzval >= length(I)). Input array klasttouch, workspace for the second stage, requires length(klasttouch) >= n. Optional input arrays csccolptr, cscrowval, and cscnzval constitute storage for the returned CSC form S. If necessary, these are resized automatically to satisfy length(csccolptr) = n + 1, length(cscrowval) = nnz(S) and length(cscnzval) = nnz(S); hence, if nnz(S) is unknown at the outset, passing in empty vectors of the appropriate type (Vector{Ti}() and Vector{Tv}() respectively) suffices, or calling the sparse! method neglecting cscrowval and cscnzval.

On return, csrrowptr, csrcolval, and csrnzval contain an unsorted-column representation of the result's transpose.

You may reuse the input arrays' storage (I, J, V) for the output arrays (csccolptr, cscrowval, cscnzval). For example, you may call sparse!(I, J, V, csrrowptr, csrcolval, csrnzval, I, J, V). Note that they will be resized to satisfy the conditions above.

For the sake of efficiency, this method performs no argument checking beyond 1 <= I[k] <= m and 1 <= J[k] <= n. Use with care. Testing with --check-bounds=yes is wise.

This method runs in O(m, n, length(I)) time. The HALFPERM algorithm described in F. Gustavson, "Two fast algorithms for sparse matrices: multiplication and permuted transposition," ACM TOMS 4(3), 250-269 (1978) inspired this method's use of a pair of counting sorts.

```
SparseArrays.sparse!(I, J, V, [m, n, combine]) -> SparseMatrixCSC
```

Variant of `sparse!` that re-uses the input vectors (`I`, `J`, `V`) for the final matrix storage. After construction the input vectors will alias the matrix buffers; `S.colptr === I`, `S.rowval === J`, and `S.nzval === V` holds, and they will be `resize!`d as necessary.

Note that some work buffers will still be allocated. Specifically, this method is a convenience wrapper around `sparse!(I, J, V, m, n, combine, klasttouch, csrrowptr, csrcolval, csrnzval, csccolptr, cscrowval, cscnzval)` where this method allocates `klasttouch`, `csrrowptr`, `csrcolval`, and `csrnzval` of appropriate size, but reuses `I`, `J`, and `V` for `csccolptr`, `cscrowval`, and `cscnzval`.

Arguments `m`, `n`, and `combine` defaults to `maximum(I)`, `maximum(J)`, and `+`, respectively.

Julia 1.10

This method requires Julia version 1.10 or later.

`SparseArrays.sparsevec` – Function.

```
sparsevec(I, V, [m, combine])
```

Create a sparse vector `S` of length `m` such that `S[I[k]] = V[k]`. Duplicates are combined using the `combine` function, which defaults to `+` if no `combine` argument is provided, unless the elements of `V` are Booleans in which case `combine` defaults to `|`.

Examples

```
julia> II = [1, 3, 3, 5]; V = [0.1, 0.2, 0.3, 0.2];

julia> sparsevec(II, V)
5-element SparseVector{Float64, Int64} with 3 stored entries:
  [1]  =  0.1
  [3]  =  0.5
  [5]  =  0.2

julia> sparsevec(II, V, 8, -)
8-element SparseVector{Float64, Int64} with 3 stored entries:
  [1]  =  0.1
  [3]  =  -0.1
  [5]  =  0.2

julia> sparsevec([1, 3, 1, 2, 2], [true, true, false, false, false])
3-element SparseVector{Bool, Int64} with 3 stored entries:
  [1]  =  1
  [2]  =  0
  [3]  =  1
```

```
sparsevec(d::Dict, [m])
```

Create a sparse vector of length `m` where the nonzero indices are keys from the dictionary, and the nonzero values are the values from the dictionary.

Examples

```
julia> sparsevec(Dict(1 => 3, 2 => 2))
2-element SparseVector{Int64, Int64} with 2 stored entries:
  [1]  =  3
  [2]  =  2
```

```
sparsevec(A)
```

Convert a vector `A` into a sparse vector of length `m`. Numerical zeros in `A` are turned into structural zeros.

Examples

```
julia> sparsevec([1.0, 2.0, 0.0, 0.0, 3.0, 0.0])
6-element SparseVector{Float64, Int64} with 3 stored entries:
  [1]  =  1.0
  [2]  =  2.0
  [5]  =  3.0
```

`Base.similar` – Method.

```
similar(A::AbstractSparseMatrixCSC{Tv,Ti}, [::Type{TvNew}, ::Type{TiNew}, m::Integer,
↪  n::Integer]) where {Tv,Ti}
```

Create an uninitialized mutable array with the given element type, index type, and size, based upon the given source `SparseMatrixCSC`. The new sparse matrix maintains the structure of the original sparse matrix, except in the case where dimensions of the output matrix are different from the output.

The output matrix has zeros in the same locations as the input, but uninitialized values for the nonzero locations.

`SparseArrays.issparse` – Function.

```
issparse(S)
```

Returns `true` if `S` is sparse, and `false` otherwise.

Examples

```
julia> sv = sparsevec([1, 4], [2.3, 2.2], 10)
10-element SparseVector{Float64, Int64} with 2 stored entries:
  [1]  =  2.3
  [4]  =  2.2

julia> issparse(sv)
true

julia> issparse(Array(sv))
false
```

`SparseArrays.nnz` – Function.

```
nnz(A)
```

Returns the number of stored (filled) elements in a sparse array.

Examples

```
julia> A = sparse(2I, 3, 3)
3×3 SparseMatrixCSC{Int64, Int64} with 3 stored entries:
 2  ·  ·
 ·  2  ·
 ·  ·  2

julia> nnz(A)
3
```

SparseArrays.findnz – Function.

```
findnz(A::SparseMatrixCSC)
```

Return a tuple `(I, J, V)` where `I` and `J` are the row and column indices of the stored ("structurally non-zero") values in sparse matrix `A`, and `V` is a vector of the values.

Examples

```
julia> A = sparse([1 2 0; 0 0 3; 0 4 0])
3×3 SparseMatrixCSC{Int64, Int64} with 4 stored entries:
 1  2  ·
 ·  ·  3
 ·  4  ·

julia> findnz(A)
([1, 1, 3, 2], [1, 2, 2, 3], [1, 2, 4, 3])
```

SparseArrays.spzeros – Function.

```
spzeros([type,]m[,n])
```

Create a sparse vector of length `m` or sparse matrix of size `m x n`. This sparse array will not contain any nonzero values. No storage will be allocated for nonzero values during construction. The type defaults to `Float64` if not specified.

Examples

```
julia> spzeros(3, 3)
3×3 SparseMatrixCSC{Float64, Int64} with 0 stored entries:
  ·    ·    ·
  ·    ·    ·
  ·    ·    ·

julia> spzeros(Float32, 4)
4-element SparseVector{Float32, Int64} with 0 stored entries
```

```
spzeros([type], I::AbstractVector, J::AbstractVector, [m, n])
```

Create a sparse matrix `S` of dimensions `m x n` with structural zeros at `S[I[k], J[k]]`.

This method can be used to construct the sparsity pattern of the matrix, and is more efficient than using e.g. `sparse(I, J, zeros(length(I)))`.

For additional documentation and an expert driver, see `SparseArrays.spzeros!`.

Julia 1.10

This methods requires Julia version 1.10 or later.

SparseArrays.spzeros! – Function.

```
spzeros!(::Type{Tv}, I::AbstractVector{Ti}, J::AbstractVector{Ti}, m::Integer, n::Integer,
         klasttouch::Vector{Ti}, csrrowptr::Vector{Ti}, csrcolval::Vector{Ti},
         [csccolptr::Vector{Ti}], [cscrowval::Vector{Ti}, cscnzval::Vector{Tv}]) where
         ↪  {Tv,Ti<:Integer}
```

Parent of and expert driver for `spzeros(I, J)` allowing user to provide preallocated storage for intermediate objects. This method is to `spzeros` what `SparseArrays.sparse!` is to `sparse`. See documentation for `SparseArrays.sparse!` for details and required buffer lengths.

Julia 1.10

This methods requires Julia version 1.10 or later.

```
SparseArrays.spzeros!(::Type{Tv}, I, J, [m, n]) -> SparseMatrixCSC{Tv}
```

Variant of `spzeros!` that re-uses the input vectors `I` and `J` for the final matrix storage. After construction the input vectors will alias the matrix buffers; `S.colptr === I` and `S.rowval === J` holds, and they will be `resize!`d as necessary.

Note that some work buffers will still be allocated. Specifically, this method is a convenience wrapper around `spzeros!(Tv, I, J, m, n, klasttouch, csrrowptr, csrcolval, csccolptr, cscrowval)` where this method allocates `klasttouch`, `csrrowptr`, and `csrcolval` of appropriate size, but reuses `I` and `J` for `csccolptr` and `cscrowval`.

Arguments `m` and `n` defaults to `maximum(I)` and `maximum(J)`.

Julia 1.10

This method requires Julia version 1.10 or later.

SparseArrays.spdiagm – Function.

```
spdiagm(kv::Pair{<:Integer,<:AbstractVector}...)
spdiagm(m::Integer, n::Integer, kv::Pair{<:Integer,<:AbstractVector}...)
```

Construct a sparse diagonal matrix from `Pair`s of vectors and diagonals. Each vector `kv.second` will be placed on the `kv.first` diagonal. By default, the matrix is square and its size is inferred from `kv`, but a non-square size `m×n` (padded with zeros as needed) can be specified by passing `m,n` as the first arguments.

Examples

```
julia> spdiagm(-1 => [1,2,3,4], 1 => [4,3,2,1])
5×5 SparseMatrixCSC{Int64, Int64} with 8 stored entries:
 ·  4  ·  ·  ·
 1  ·  3  ·  ·
 ·  2  ·  2  ·
```

```
·  ·  3  ·  1
·  ·  ·  4  ·
```

```
spdiagm(v::AbstractVector)
spdiagm(m::Integer, n::Integer, v::AbstractVector)
```

Construct a sparse matrix with elements of the vector as diagonal elements. By default (no given `m` and `n`), the matrix is square and its size is given by `length(v)`, but a non-square size `m×n` can be specified by passing `m` and `n` as the first arguments.

Julia 1.6

These functions require at least Julia 1.6.

Examples

```
julia> spdiagm([1,2,3])
3×3 SparseMatrixCSC{Int64, Int64} with 3 stored entries:
 1  ·  ·
 ·  2  ·
 ·  ·  3

julia> spdiagm(sparse([1,0,3]))
3×3 SparseMatrixCSC{Int64, Int64} with 2 stored entries:
 1  ·  ·
 ·  ·  ·
 ·  ·  3
```

`SparseArrays.sparse_hcat` – Function.

```
sparse_hcat(A...)
```

Concatenate along dimension 2. Return a SparseMatrixCSC object.

Julia 1.8

This method was added in Julia 1.8. It mimics previous concatenation behavior, where the concatenation with specialized "sparse" matrix types from LinearAlgebra.jl automatically yielded sparse output even in the absence of any SparseArray argument.

`SparseArrays.sparse_vcat` – Function.

```
sparse_vcat(A...)
```

Concatenate along dimension 1. Return a SparseMatrixCSC object.

Julia 1.8

This method was added in Julia 1.8. It mimics previous concatenation behavior, where the concatenation with specialized "sparse" matrix types from LinearAlgebra.jl automatically yielded sparse output even in the absence of any SparseArray argument.

SparseArrays.sparse_hvcat – Function.

```
sparse_hvcat(rows::Tuple{Vararg{Int}}, values...)
```

Sparse horizontal and vertical concatenation in one call. This function is called for block matrix syntax. The first argument specifies the number of arguments to concatenate in each block row.

Julia 1.8

This method was added in Julia 1.8. It mimics previous concatenation behavior, where the concatenation with specialized "sparse" matrix types from LinearAlgebra.jl automatically yielded sparse output even in the absence of any SparseArray argument.

SparseArrays.blockdiag – Function.

```
blockdiag(A...)
```

Concatenate matrices block-diagonally. Currently only implemented for sparse matrices.

Examples

```
julia> blockdiag(sparse(2I, 3, 3), sparse(4I, 2, 2))
5×5 SparseMatrixCSC{Int64, Int64} with 5 stored entries:
 2  ·  ·  ·  ·
 ·  2  ·  ·  ·
 ·  ·  2  ·  ·
 ·  ·  ·  4  ·
 ·  ·  ·  ·  4
```

SparseArrays.sprand – Function.

```
sprand([rng],[T::Type],m,[n],p::AbstractFloat)
sprand([rng],m,[n],p::AbstractFloat,[rfn=rand])
```

Create a random length `m` sparse vector or `m` by `n` sparse matrix, in which the probability of any element being nonzero is independently given by `p` (and hence the mean density of nonzeros is also exactly `p`). The optional `rng` argument specifies a random number generator, see Random Numbers. The optional `T` argument specifies the element type, which defaults to `Float64`.

By default, nonzero values are sampled from a uniform distribution using the `rand` function, i.e. by `rand(T)`, or `rand(rng, T)` if `rng` is supplied; for the default `T=Float64`, this corresponds to nonzero values sampled uniformly in `[0,1)`.

You can sample nonzero values from a different distribution by passing a custom `rfn` function instead of `rand`. This should be a function `rfn(k)` that returns an array of `k` random numbers sampled from the desired distribution; alternatively, if `rng` is supplied, it should instead be a function `rfn(rng, k)`.

Examples

```
julia> sprand(Bool, 2, 2, 0.5)
2×2 SparseMatrixCSC{Bool, Int64} with 2 stored entries:
 1  1
 ·  ·
```

```
julia> sprand(Float64, 3, 0.75)
3-element SparseVector{Float64, Int64} with 2 stored entries:
  [1]  =  0.795547
  [2]  =  0.49425
```

SparseArrays.sprandn – Function.

```
sprandn([rng][,Type],m[,n],p::AbstractFloat)
```

Create a random sparse vector of length m or sparse matrix of size m by n with the specified (independent) probability p of any entry being nonzero, where nonzero values are sampled from the normal distribution. The optional rng argument specifies a random number generator, see Random Numbers.

Julia 1.1

Specifying the output element type Type requires at least Julia 1.1.

Examples

```
julia> sprandn(2, 2, 0.75)
2×2 SparseMatrixCSC{Float64, Int64} with 3 stored entries:
 -1.20577     ·
  0.311817  -0.234641
```

SparseArrays.nonzeros – Function.

```
nonzeros(A)
```

Return a vector of the structural nonzero values in sparse array A. This includes zeros that are explicitly stored in the sparse array. The returned vector points directly to the internal nonzero storage of A, and any modifications to the returned vector will mutate A as well. See rowvals and nzrange.

Examples

```
julia> A = sparse(2I, 3, 3)
3×3 SparseMatrixCSC{Int64, Int64} with 3 stored entries:
 2  ·  ·
 ·  2  ·
 ·  ·  2

julia> nonzeros(A)
3-element Vector{Int64}:
 2
 2
 2
```

SparseArrays.rowvals – Function.

```
rowvals(A)
```

Return a vector of the row indices of sparse array A. Any modifications to the returned vector will mutate A as well. Providing access to how the row indices are stored internally can be useful in conjunction with iterating over structural nonzero values. See also nonzeros and nzrange.

Examples

```
julia> A = sparse(2I, 3, 3)
3×3 SparseMatrixCSC{Int64, Int64} with 3 stored entries:
 2  ·  ·
 ·  2  ·
 ·  ·  2

julia> rowvals(A)
3-element Vector{Int64}:
 1
 2
 3
```

SparseArrays.nzrange – Function.

```
nzrange(A, col::Integer)
```

Return the range of indices to the structural nonzero values of column `col` of sparse array `A`. In conjunction with `nonzeros` and `rowvals`, this allows for convenient iterating over a sparse matrix :

```
A = sparse(I,J,V)
rows = rowvals(A)
vals = nonzeros(A)
m, n = size(A)
for j = 1:n
   for i in nzrange(A, j)
      row = rows[i]
      val = vals[i]
      # perform sparse wizardry...
   end
end
```

Warning

Adding or removing nonzero elements to the matrix may invalidate the `nzrange`, one should not mutate the matrix while iterating.

```
nzrange(x::SparseVectorUnion, col)
```

Give the range of indices to the structural nonzero values of a sparse vector. The column index `col` is ignored (assumed to be 1).

SparseArrays.droptol! – Function.

```
droptol!(A::AbstractSparseMatrixCSC, tol)
```

Removes stored values from `A` whose absolute value is less than or equal to `tol`.

```
droptol!(x::AbstractCompressedVector, tol)
```

Removes stored values from `x` whose absolute value is less than or equal to `tol`.

SparseArrays.dropzeros! – Function.

```
dropzeros!(x::AbstractCompressedVector)
```

Removes stored numerical zeros from x.

For an out-of-place version, see dropzeros. For algorithmic information, see fkeep!.

SparseArrays.dropzeros – Function.

```
dropzeros(A::AbstractSparseMatrixCSC;)
```

Generates a copy of A and removes stored numerical zeros from that copy.

For an in-place version and algorithmic information, see dropzeros!.

Examples

```
julia> A = sparse([1, 2, 3], [1, 2, 3], [1.0, 0.0, 1.0])
3×3 SparseMatrixCSC{Float64, Int64} with 3 stored entries:
 1.0   ·    ·
  ·   0.0   ·
  ·    ·   1.0

julia> dropzeros(A)
3×3 SparseMatrixCSC{Float64, Int64} with 2 stored entries:
 1.0   ·    ·
  ·    ·    ·
  ·    ·   1.0
```

```
dropzeros(x::AbstractCompressedVector)
```

Generates a copy of x and removes numerical zeros from that copy.

For an in-place version and algorithmic information, see dropzeros!.

Examples

```
julia> A = sparsevec([1, 2, 3], [1.0, 0.0, 1.0])
3-element SparseVector{Float64, Int64} with 3 stored entries:
  [1]  =  1.0
  [2]  =  0.0
  [3]  =  1.0

julia> dropzeros(A)
3-element SparseVector{Float64, Int64} with 2 stored entries:
  [1]  =  1.0
  [3]  =  1.0
```

SparseArrays.permute – Function.

```
permute(A::AbstractSparseMatrixCSC{Tv,Ti}, p::AbstractVector{<:Integer},
        q::AbstractVector{<:Integer}) where {Tv,Ti}
```

Bilaterally permute A, returning PAQ (A[p,q]). Column-permutation q's length must match A's column count (length(q) == size(A, 2)). Row-permutation p's length must match A's row count (length(p) == size(A, 1)).

For expert drivers and additional information, see permute!.

Examples

```
julia> A = spdiagm(0 => [1, 2, 3, 4], 1 => [5, 6, 7])
4×4 SparseMatrixCSC{Int64, Int64} with 7 stored entries:
 1  5  ·  ·
 ·  2  6  ·
 ·  ·  3  7
 ·  ·  ·  4

julia> permute(A, [4, 3, 2, 1], [1, 2, 3, 4])
4×4 SparseMatrixCSC{Int64, Int64} with 7 stored entries:
 ·  ·  ·  4
 ·  ·  3  7
 ·  2  6  ·
 1  5  ·  ·

julia> permute(A, [1, 2, 3, 4], [4, 3, 2, 1])
4×4 SparseMatrixCSC{Int64, Int64} with 7 stored entries:
 ·  ·  5  1
 ·  6  2  ·
 7  3  ·  ·
 4  ·  ·  ·
```

Base.permute! – Method.

```
permute!(X::AbstractSparseMatrixCSC{Tv,Ti}, A::AbstractSparseMatrixCSC{Tv,Ti},
         p::AbstractVector{<:Integer}, q::AbstractVector{<:Integer},
         [C::AbstractSparseMatrixCSC{Tv,Ti}]) where {Tv,Ti}
```

Bilaterally permute A, storing result PAQ (A[p,q]) in X. Stores intermediate result (AQ)^T (transpose(A[:,q])) in optional argument C if present. Requires that none of X, A, and, if present, C alias each other; to store result PAQ back into A, use the following method lacking X:

```
permute!(A::AbstractSparseMatrixCSC{Tv,Ti}, p::AbstractVector{<:Integer},
         q::AbstractVector{<:Integer}[, C::AbstractSparseMatrixCSC{Tv,Ti},
         [workcolptr::Vector{Ti}]]) where {Tv,Ti}
```

X's dimensions must match those of A (size(X, 1) == size(A, 1) and size(X, 2) == size(A, 2)), and X must have enough storage to accommodate all allocated entries in A (length(rowvals(X)) >= nnz(A) and length(nonzeros(X)) >= nnz(A)). Column-permutation q's length must match A's column count (length(q) == size(A, 2)). Row-permutation p's length must match A's row count (length(p) == size(A, 1)).

C's dimensions must match those of transpose(A) (size(C, 1) == size(A, 2) and size(C, 2) == size(A, 1)), and C must have enough storage to accommodate all allocated entries in A (length(rowvals(C)) >= nnz(A) and length(nonzeros(C)) >= nnz(A)).

For additional (algorithmic) information, and for versions of these methods that forgo argument checking, see (unexported) parent methods unchecked_noalias_permute! and unchecked_aliasing_permute!.

See also permute.

SparseArrays.halfperm! – Function.

```
halfperm!(X::AbstractSparseMatrixCSC{Tv,Ti}, A::AbstractSparseMatrixCSC{TvA,Ti},
          q::AbstractVector{<:Integer}, f::Function = identity) where {Tv,TvA,Ti}
```

Column-permute and transpose A, simultaneously applying f to each entry of A, storing the result (f(A)Q)^T (map(f, transpose(A[:,q]))) in X.

Element type Tv of X must match f(::TvA), where TvA is the element type of A. X's dimensions must match those of transpose(A) (size(X, 1) == size(A, 2) and size(X, 2) == size(A, 1)), and X must have enough storage to accommodate all allocated entries in A (length(rowvals(X)) >= nnz(A) and length(nonzeros(X)) >= nnz(A)). Column-permutation q's length must match A's column count (length(q) == size(A, 2)).

This method is the parent of several methods performing transposition and permutation operations on SparseMatrixCSCs. As this method performs no argument checking, prefer the safer child methods ([c]transpose[!], permute[!]) to direct use.

This method implements the HALFPERM algorithm described in F. Gustavson, "Two fast algorithms for sparse matrices: multiplication and permuted transposition," ACM TOMS 4(3), 250-269 (1978). The algorithm runs in O(size(A, 1), size(A, 2), nnz(A)) time and requires no space beyond that passed in.

SparseArrays.ftranspose! – Function.

```
ftranspose!(X::AbstractSparseMatrixCSC{Tv,Ti}, A::AbstractSparseMatrixCSC{Tv,Ti}, f::Function)
↪  where {Tv,Ti}
```

Transpose A and store it in X while applying the function f to the non-zero elements. Does not remove the zeros created by f. size(X) must be equal to size(transpose(A)). No additional memory is allocated other than resizing the rowval and nzval of X, if needed.

See halfperm!

Chapter 100

Noteworthy External Sparse Packages

Several other Julia packages provide sparse matrix implementations that should be mentioned:

1. SuiteSparseGraphBLAS.jl is a wrapper over the fast, multithreaded SuiteSparse:GraphBLAS C library. On CPU this is typically the fastest option, often significantly outperforming MKLSparse.
2. CUDA.jl exposes the CUSPARSE library for GPU sparse matrix operations.
3. SparseMatricesCSR.jl provides a Julia native implementation of the Compressed Sparse Rows (CSR) format.
4. MKLSparse.jl accelerates SparseArrays sparse-dense matrix operations using Intel's MKL library.
5. SparseArrayKit.jl available for multidimensional sparse arrays.
6. LuxurySparse.jl provides static sparse array formats, as well as a coordinate format.
7. ExtendableSparse.jl enables fast insertion into sparse matrices using a lazy approach to new stored indices.
8. Finch.jl supports extensive multidimensional sparse array formats and operations through a mini tensor language and compiler, all in native Julia. Support for COO, CSF, CSR, CSC and more, as well as operations like broadcast, reduce, etc. and custom operations.

External packages providing sparse direct solvers:

1. KLU.jl
2. Pardiso.jl

External packages providing solvers for iterative solution of eigensystems and singular value decompositions:

1. ArnoldiMethods.jl
2. KrylovKit
3. Arpack.jl

External packages for working with graphs:

1. Graphs.jl

Chapter 101

Statistics

The Statistics standard library module contains basic statistics functionality.

`Statistics.std` – Function.

```
std(itr; corrected::Bool=true, mean=nothing[, dims])
```

Compute the sample standard deviation of collection `itr`.

The algorithm returns an estimator of the generative distribution's standard deviation under the assumption that each entry of `itr` is a sample drawn from the same unknown distribution, with the samples uncorrelated. For arrays, this computation is equivalent to calculating `sqrt.(sum(abs2.(itr .- mean(itr))) / (length(itr) - 1))`. If `corrected` is `true`, then the sum is scaled with `n-1`, whereas the sum is scaled with `n` if `corrected` is `false` with `n` the number of elements in `itr`.

If `itr` is an `AbstractArray`, `dims` can be provided to compute the standard deviation over dimensions.

A pre-computed `mean` may be provided. When `dims` is specified, `mean` must be an array with the same shape as `mean(itr, dims=dims)` (additional trailing singleton dimensions are allowed).

Note

If array contains NaN or `missing` values, the result is also NaN or `missing` (`missing` takes precedence if array contains both). Use the `skipmissing` function to omit `missing` entries and compute the standard deviation of non-missing values.

`Statistics.stdm` – Function.

```
stdm(itr, mean; corrected::Bool=true[, dims])
```

Compute the sample standard deviation of collection `itr`, with known mean(s) `mean`.

The algorithm returns an estimator of the generative distribution's standard deviation under the assumption that each entry of `itr` is a sample drawn from the same unknown distribution, with the samples uncorrelated. For arrays, this computation is equivalent to calculating `sqrt.(sum(abs2.(itr .- mean(itr))) / (length(itr) - 1))`. If `corrected` is `true`, then the sum is scaled with `n-1`, whereas the sum is scaled with `n` if `corrected` is `false` with `n` the number of elements in `itr`.

If `itr` is an `AbstractArray`, `dims` can be provided to compute the standard deviation over dimensions. In that case, `mean` must be an array with the same shape as `mean(itr, dims=dims)` (additional trailing singleton dimensions are allowed).

Note

If array contains NaN or `missing` values, the result is also NaN or `missing` (`missing` takes precedence if array contains both). Use the `skipmissing` function to omit `missing` entries and compute the standard deviation of non-missing values.

`Statistics.var` – Function.

```
var(itr; corrected::Bool=true, mean=nothing[, dims])
```

Compute the sample variance of collection `itr`.

The algorithm returns an estimator of the generative distribution's variance under the assumption that each entry of `itr` is a sample drawn from the same unknown distribution, with the samples uncorrelated. For arrays, this computation is equivalent to calculating `sum(abs2.(itr .- mean(itr))) / (length(itr) - 1)`. If `corrected` is `true`, then the sum is scaled with `n-1`, whereas the sum is scaled with `n` if `corrected` is `false` where `n` is the number of elements in `itr`.

If `itr` is an `AbstractArray`, `dims` can be provided to compute the variance over dimensions.

A pre-computed `mean` may be provided. When `dims` is specified, `mean` must be an array with the same shape as `mean(itr, dims=dims)` (additional trailing singleton dimensions are allowed).

Note

If array contains NaN or `missing` values, the result is also NaN or `missing` (`missing` takes precedence if array contains both). Use the `skipmissing` function to omit `missing` entries and compute the variance of non-missing values.

`Statistics.varm` – Function.

```
varm(itr, mean; dims, corrected::Bool=true)
```

Compute the sample variance of collection `itr`, with known mean(s) `mean`.

The algorithm returns an estimator of the generative distribution's variance under the assumption that each entry of `itr` is a sample drawn from the same unknown distribution, with the samples uncorrelated. For arrays, this computation is equivalent to calculating `sum(abs2.(itr .- mean(itr))) / (length(itr) - 1)`. If `corrected` is `true`, then the sum is scaled with `n-1`, whereas the sum is scaled with `n` if `corrected` is `false` with `n` the number of elements in `itr`.

If `itr` is an `AbstractArray`, `dims` can be provided to compute the variance over dimensions. In that case, `mean` must be an array with the same shape as `mean(itr, dims=dims)` (additional trailing singleton dimensions are allowed).

Note

If array contains NaN or `missing` values, the result is also NaN or `missing` (`missing` takes precedence if array contains both). Use the `skipmissing` function to omit `missing` entries and compute the variance of non-missing values.

`Statistics.cor` – Function.

```
cor(x::AbstractVector)
```

Return the number one.

```
cor(X::AbstractMatrix; dims::Int=1)
```

Compute the Pearson correlation matrix of the matrix X along the dimension `dims`.

```
cor(x::AbstractVector, y::AbstractVector)
```

Compute the Pearson correlation between the vectors x and y.

```
cor(X::AbstractVecOrMat, Y::AbstractVecOrMat; dims=1)
```

Compute the Pearson correlation between the vectors or matrices X and Y along the dimension `dims`.

`Statistics.cov` – Function.

```
cov(x::AbstractVector; corrected::Bool=true)
```

Compute the variance of the vector `x`. If `corrected` is `true` (the default) then the sum is scaled with `n-1`, whereas the sum is scaled with `n` if `corrected` is `false` where `n = length(x)`.

```
cov(X::AbstractMatrix; dims::Int=1, corrected::Bool=true)
```

Compute the covariance matrix of the matrix X along the dimension `dims`. If `corrected` is `true` (the default) then the sum is scaled with `n-1`, whereas the sum is scaled with `n` if `corrected` is `false` where `n = size(X, dims)`.

```
cov(x::AbstractVector, y::AbstractVector; corrected::Bool=true)
```

Compute the covariance between the vectors `x` and `y`. If `corrected` is `true` (the default), computes $\frac{1}{n-1}\sum_{i=1}^{n}(x_i - \bar{x})(y_i - \bar{y})^*$ where $*$ denotes the complex conjugate and `n = length(x) = length(y)`. If `corrected` is `false`, computes $\frac{1}{n}\sum_{i=1}^{n}(x_i - \bar{x})(y_i - \bar{y})^*$.

```
cov(X::AbstractVecOrMat, Y::AbstractVecOrMat; dims::Int=1, corrected::Bool=true)
```

Compute the covariance between the vectors or matrices X and Y along the dimension `dims`. If `corrected` is `true` (the default) then the sum is scaled with `n-1`, whereas the sum is scaled with `n` if `corrected` is `false` where `n = size(X, dims) = size(Y, dims)`.

`Statistics.mean!` – Function.

```
mean!(r, v)
```

Compute the mean of v over the singleton dimensions of `r`, and write results to `r`. Note that the target must not alias with the source.

Examples

```
julia> using Statistics

julia> v = [1 2; 3 4]
2×2 Matrix{Int64}:
 1  2
 3  4

julia> mean!([1., 1.], v)
2-element Vector{Float64}:
 1.5
 3.5

julia> mean!([1. 1.], v)
1×2 Matrix{Float64}:
 2.0  3.0
```

`Statistics.mean` – Function.

```
mean(itr)
```

Compute the mean of all elements in a collection.

Note

If `itr` contains `NaN` or `missing` values, the result is also `NaN` or `missing` (`missing` takes precedence if array contains both). Use the `skipmissing` function to omit `missing` entries and compute the mean of non-missing values.

Examples

```
julia> using Statistics

julia> mean(1:20)
10.5

julia> mean([1, missing, 3])
missing

julia> mean(skipmissing([1, missing, 3]))
2.0
```

```
mean(f, itr)
```

Apply the function `f` to each element of collection `itr` and take the mean.

```
julia> using Statistics

julia> mean(√, [1, 2, 3])
1.3820881233139908

julia> mean([√1, √2, √3])
1.3820881233139908
```

```
mean(f, A::AbstractArray; dims)
```

Apply the function `f` to each element of array `A` and take the mean over dimensions `dims`.

Julia 1.3

This method requires at least Julia 1.3.

```
julia> using Statistics

julia> mean(√, [1, 2, 3])
1.3820881233139908

julia> mean([√1, √2, √3])
1.3820881233139908

julia> mean(√, [1 2 3; 4 5 6], dims=2)
2×1 Matrix{Float64}:
 1.3820881233139908
 2.2285192400943226
```

```
mean(A::AbstractArray; dims)
```

Compute the mean of an array over the given dimensions.

Julia 1.1

`mean` for empty arrays requires at least Julia 1.1.

Examples

```
julia> using Statistics

julia> A = [1 2; 3 4]
2×2 Matrix{Int64}:
 1  2
 3  4

julia> mean(A, dims=1)
1×2 Matrix{Float64}:
 2.0  3.0

julia> mean(A, dims=2)
2×1 Matrix{Float64}:
 1.5
 3.5
```

`Statistics.median!` – Function.

```
median!(v)
```

Like `median`, but may overwrite the input vector.

`Statistics.median` – Function.

```
median(itr)
```

Compute the median of all elements in a collection. For an even number of elements no exact median element exists, so the result is equivalent to calculating mean of two median elements.

Note

If `itr` contains NaN or `missing` values, the result is also NaN or `missing` (`missing` takes precedence if `itr` contains both). Use the `skipmissing` function to omit `missing` entries and compute the median of non-missing values.

Examples

```
julia> using Statistics

julia> median([1, 2, 3])
2.0

julia> median([1, 2, 3, 4])
2.5

julia> median([1, 2, missing, 4])
missing

julia> median(skipmissing([1, 2, missing, 4]))
2.0
```

```
median(A::AbstractArray; dims)
```

Compute the median of an array along the given dimensions.

Examples

```
julia> using Statistics

julia> median([1 2; 3 4], dims=1)
1×2 Matrix{Float64}:
 2.0  3.0
```

```
median(f, v)
```

Apply the function `f` to each element of collection `v` and then compute the median.

```
julia> using Statistics

julia> median(√, [1, 3, 2])
1.4142135623730951

julia> median([√1, √3, √2])
1.4142135623730951
```

`Statistics.middle` – Function.

```
middle(x)
```

Compute the middle of a scalar value, which is equivalent to x itself, but of the type of `middle(x, x)` for consistency.

```
middle(x, y)
```

Compute the middle of two numbers x and y, which is equivalent in both value and type to computing their mean (`(x + y) / 2`).

```
middle(a::AbstractArray)
```

Compute the middle of an array a, which consists of finding its extrema and then computing their mean.

```
julia> using Statistics

julia> middle(1:10)
5.5

julia> a = [1,2,3.6,10.9]
4-element Vector{Float64}:
  1.0
  2.0
  3.6
 10.9

julia> middle(a)
5.95
```

`Statistics.quantile!` – Function.

```
quantile!([q::AbstractArray, ] v::AbstractVector, p; sorted=false, alpha::Real=1.0,
↪  beta::Real=alpha)
```

Compute the quantile(s) of a vector v at a specified probability or vector or tuple of probabilities p on the interval [0,1]. If p is a vector, an optional output array q may also be specified. (If not provided, a new output array is created.) The keyword argument `sorted` indicates whether v can be assumed to be sorted; if `false` (the default), then the elements of v will be partially sorted in-place.

Samples quantile are defined by `Q(p) = (1-γ)*x[j] + γ*x[j+1]`, where `x[j]` is the j-th order statistic of v, `j = floor(n*p + m)`, `m = alpha + p*(1 - alpha - beta)` and `γ = n*p + m - j`.

By default (`alpha = beta = 1`), quantiles are computed via linear interpolation between the points `((k-1)/(n-1), x[k])`, for `k = 1:n` where `n = length(v)`. This corresponds to Definition 7 of Hyndman and Fan (1996), and is the same as the R and NumPy default.

The keyword arguments `alpha` and `beta` correspond to the same parameters in Hyndman and Fan, setting them to different values allows to calculate quantiles with any of the methods 4-9 defined in this paper:

- Def. 4: `alpha=0`, `beta=1`

- Def. 5: `alpha=0.5`, `beta=0.5` (MATLAB default)
- Def. 6: `alpha=0`, `beta=0` (Excel `PERCENTILE.EXC`, Python default, Stata `altdef`)
- Def. 7: `alpha=1`, `beta=1` (Julia, R and NumPy default, Excel `PERCENTILE` and `PERCENTILE.INC`, Python `'inclusive'`)
- Def. 8: `alpha=1/3`, `beta=1/3`
- Def. 9: `alpha=3/8`, `beta=3/8`

Note

An `ArgumentError` is thrown if v contains `NaN` or `missing` values.

References

- Hyndman, R.J and Fan, Y. (1996) "Sample Quantiles in Statistical Packages", *The American Statistician*, Vol. 50, No. 4, pp. 361-365
- Quantile on Wikipedia details the different quantile definitions

Examples

```
julia> using Statistics

julia> x = [3, 2, 1];

julia> quantile!(x, 0.5)
2.0

julia> x
3-element Vector{Int64}:
 1
 2
 3

julia> y = zeros(3);

julia> quantile!(y, x, [0.1, 0.5, 0.9]) === y
true

julia> y
3-element Vector{Float64}:
 1.2
 2.0
 2.8
```

`Statistics.quantile` – Function.

```
quantile(itr, p; sorted=false, alpha::Real=1.0, beta::Real=alpha)
```

Compute the quantile(s) of a collection `itr` at a specified probability or vector or tuple of probabilities `p` on the interval [0,1]. The keyword argument `sorted` indicates whether `itr` can be assumed to be sorted.

Samples quantile are defined by `Q(p) = (1-γ)*x[j] + γ*x[j+1]`, where `x[j]` is the j-th order statistic of `itr`, `j = floor(n*p + m)`, `m = alpha + p*(1 - alpha - beta)` and `γ = n*p + m - j`.

By default (`alpha = beta = 1`), quantiles are computed via linear interpolation between the points `((k-1)/(n-1), x[k])`, for `k = 1:n` where `n = length(itr)`. This corresponds to Definition 7 of Hyndman and Fan (1996), and is the same as the R and NumPy default.

The keyword arguments `alpha` and `beta` correspond to the same parameters in Hyndman and Fan, setting them to different values allows to calculate quantiles with any of the methods 4-9 defined in this paper:

- Def. 4: `alpha=0`, `beta=1`
- Def. 5: `alpha=0.5`, `beta=0.5` (MATLAB default)
- Def. 6: `alpha=0`, `beta=0` (Excel `PERCENTILE.EXC`, Python default, Stata `altdef`)
- Def. 7: `alpha=1`, `beta=1` (Julia, R and NumPy default, Excel `PERCENTILE` and `PERCENTILE.INC`, Python `'inclusive'`)
- Def. 8: `alpha=1/3`, `beta=1/3`
- Def. 9: `alpha=3/8`, `beta=3/8`

Note

An `ArgumentError` is thrown if v contains NaN or `missing` values. Use the `skipmissing` function to omit `missing` entries and compute the quantiles of non-missing values.

References

- Hyndman, R.J and Fan, Y. (1996) "Sample Quantiles in Statistical Packages", *The American Statistician*, Vol. 50, No. 4, pp. 361-365
- Quantile on Wikipedia details the different quantile definitions

Examples

```
julia> using Statistics

julia> quantile(0:20, 0.5)
10.0

julia> quantile(0:20, [0.1, 0.5, 0.9])
3-element Vector{Float64}:
  2.0
 10.0
 18.0

julia> quantile(skipmissing([1, 10, missing]), 0.5)
5.5
```

```
quantile(f, v)
```

Apply the function `f` to each element of collection `v` and then compute the quantile(s) at a specified probability or vector or tuple of probabilities `p` on the interval [0,1].

```
julia> using Statistics

julia> quantile(√, [1, 3, 2], 0.3)
1.248528137423857

julia> quantile([√1, √3, √2], 0.3)
1.248528137423857

julia> quantile(√, [1, 3, 2], (0.3, 0.4, 0.5))
(1.248528137423857, 1.3313708498984762, 1.4142135623730951)

julia> quantile(.√[1, 3, 2], (0.3, 0.4, 0.5))
(1.248528137423857, 1.3313708498984762, 1.4142135623730951)
```

Chapter 102

StyledStrings

Note

The API for StyledStrings and AnnotatedStrings is considered experimental and is subject to change between Julia versions.

102.1 Styling

When working with strings, formatting and styling often appear as a secondary concern.

For instance, when printing to a terminal you might want to sprinkle ANSI escape sequences in the output, when outputting HTML styling constructs (`<span style="...">`, etc.) serve a similar purpose, and so on. It is possible to simply insert the raw styling constructs into the string next to the content itself, but it quickly becomes apparent that this is not well suited for anything but the most basic use cases. Not all terminals support the same ANSI codes, the styling constructs need to be painstakingly removed when calculating the width of already-styled content, and that's before you even get into handling multiple output formats.

Instead of leaving this headache to be widely experienced downstream, it is tackled head-on by the introduction of a special string type (`AnnotatedString`). This string type wraps any other `AbstractString` type and allows for formatting information to be applied to regions (e.g. characters 1 through to 7 are bold and red).

Regions of a string are styled by applying `Face`s (think "typeface") to them — a structure that holds styling information. As a convenience, faces in the global faces dictionary (e.g. `shadow`) can just be named instead of giving the `Face` directly.

Along with these capabilities, we also provide a convenient way for constructing `AnnotatedString`s, detailed in Styled String Literals.

```
julia> using StyledStrings

julia> styled"{yellow:hello} {blue:there}"
"hello there"
```

102.2 Annotated Strings

It is sometimes useful to be able to hold metadata relating to regions of a string. A `AnnotatedString` wraps another string and allows for regions of it to be annotated with labelled values (`:label => value`). All generic string operations are applied to the underlying string. However, when possible, styling information is preserved. This means you can manipulate a `AnnotatedString` —taking substrings, padding them, concatenating them with other strings— and the metadata annotations will "come along for the ride".

This string type is fundamental to the StyledStrings stdlib, which uses `:face`-labelled annotations to hold styling information.

When concatenating a `AnnotatedString`, take care to use `annotatedstring` instead of `string` if you want to keep the string annotations.

```
julia> str = AnnotatedString("hello there", [(1:5, :word, :greeting), (7:11, :label, 1)])
"hello there"

julia> length(str)
11

julia> lpad(str, 14)
"   hello there"

julia> typeof(lpad(str, 7))
AnnotatedString{String}

julia> str2 = AnnotatedString(" julia", [(2:6, :face, :magenta)])
" julia"

julia> annotatedstring(str, str2)
"hello there julia"

julia> str * str2 == annotatedstring(str, str2) # *-concatenation works
true
```

The annotations of a `AnnotatedString` can be accessed and modified via the `annotations` and `annotate!` functions.

102.3 Styling via `AnnotatedStrings`

102.4 Faces

The Face type

A `Face` specifies details of a typeface that text can be set in. It covers a set of basic attributes that generalize well across different formats, namely:

- `font`
- `height`
- `weight`

- `slant`
- `foreground`
- `background`
- `underline`
- `strikethrough`
- `inverse`
- `inherit`

For details on the particular forms these attributes take, see the Face docstring, but of particular interest is `inherit` as it allows you to *inherit* attributes from other Faces.

The global faces dictionary

To make referring to particular styles more convenient, there is a global `Dict{Symbol, Face}` that allows for Faces to be referred to simply by name. Packages can add faces to this dictionary via the `addface!` function, and the loaded faces can be easily customized.

Appropriate face naming

Any package registering new faces should ensure that they are prefixed by the package name, i.e. follow the format `mypackage_myface`. This is important for predictability, and to prevent name clashes.

Furthermore, packages should take care to use (and introduce) *semantic* faces (like `code`) over direct colours and styles (like `cyan`). This is helpful in a number of ways, from making the intent in usage more obvious, aiding composability, and making user customisation more intuitive.

There are two set of exemptions to the package-prefix rule:

- the set of basic faces that are part of the default value of the faces dictionary
- faces introduced by Julia's own standard library, namely `JuliaSyntaxHighlighting`

Basic faces

Basic faces are intended to represent a general idea that is widely applicable.

For setting some text with a certain attribute, we have the `bold`, `light`, `italic`, `underline`, `strikethrough`, and `inverse` faces.

There are also named faces for the 16 terminal colors: `black`, `red`, `green`, `yellow`, `blue`, `magenta`, `cyan`, `white`, `bright_black`/`grey`/`gray`, `bright_red`, `bright_green`, `bright_blue`, `bright_magenta`, `bright_cyan`, and `bright_white`.

For shadowed text (i.e. dim but there) there is the `shadow` face. To indicate a selected region, there is the `region` face. Similarly for emphasis and highlighting the `emphasis` and `highlight` faces are defined. There is also `code` for code-like text.

For visually indicating the severity of messages, the `error`, `warning`, `success`, `info`, `note`, and `tip` faces are defined.

Customisation of faces (`Faces.toml`)

It is good for the name faces in the global face dictionary to be customizable. Theming and aesthetics are nice, and it is important for accessibility reasons too. A TOML file can be parsed into a list of Face specifications that are merged with the pre-existing entry in the face dictionary.

A Face is represented in TOML like so:

```
[facename]
attribute = "value"
...

[package.facename]
attribute = "value"
```

For example, if the shadow face is too hard to read it can be made brighter like so:

```
[shadow]
foreground = "white"
```

On initialization, the `config/faces.toml` file under the first Julia depot (usually `~/.julia`) is loaded.

Applying faces to a `AnnotatedString`

By convention, the `:face` attributes of a AnnotatedString hold information on the Faces that currently apply. This can be given in multiple forms, as a single `Symbol` naming a Faces in the global face dictionary, a Face itself, or a vector of either.

The `show(::IO, ::MIME"text/plain", ::AnnotatedString)` and `show(::IO, ::MIME"text/html", ::AnnotatedString)` methods both look at the `:face` attributes and merge them all together when determining the overall styling.

We can supply `:face` attributes to a `AnnotatedString` during construction, add them to the properties list afterwards, or use the convenient Styled String literals.

```
julia> str1 = AnnotatedString("blue text", [(1:9, :face, :blue)])
"blue text"

julia> str2 = styled"{blue:blue text}"
"blue text"

julia> str1 == str2
true

julia> sprint(print, str1, context = :color => true)
"\e[34mblue text\e[39m"

julia> sprint(show, MIME("text/html"), str1, context = :color => true)
"<span style=\"color: #195eb3\">blue text</span>"
```

102.5 Styled String Literals

To ease construction of `AnnotatedString`s with `Face`s applied, the `styled"..."` styled string literal allows for the content and attributes to be easily expressed together via a custom grammar.

Within a `styled"..."` literal, curly braces are considered special characters and must be escaped in normal usage (`\{`, `\}`). This allows them to be used to express annotations with (nestable) `{annotations...:text}` constructs.

The `annotations...` component is a comma-separated list of three types of annotations.

- Face names
- Inline Face expressions `(key=val,...)`
- `key=value` pairs

Interpolation is possible everywhere except for inline face keys.

For more information on the grammar, see the extended help of the `styled"..."` docstring.

As an example, we can demonstrate the list of built-in faces mentioned above like so:

```
julia> println(styled"
The basic font-style attributes are {bold:bold}, {light:light}, {italic:italic},
{underline:underline}, and {strikethrough:strikethrough}.

In terms of color, we have named faces for the 16 standard terminal colors:
 {black:■} {red:■} {green:■} {yellow:■} {blue:■} {magenta:■} {cyan:■} {white:■}
 {bright_black:■} {bright_red:■} {bright_green:■} {bright_yellow:■} {bright_blue:■}
 ↪  {bright_magenta:■} {bright_cyan:■} {bright_white:■}

Since {code:bright_black} is effectively grey, we define two aliases for it:
{code:grey} and {code:gray} to allow for regional spelling differences.

To flip the foreground and background colors of some text, you can use the
{code:inverse} face, for example: {magenta:some {inverse:inverse} text}.

The intent-based basic faces are {shadow:shadow} (for dim but visible text),
{region:region} for selections, {emphasis:emphasis}, and {highlight:highlight}.
As above, {code:code} is used for code-like text.

Lastly, we have the 'message severity' faces: {error:error}, {warning:warning},
{success:success}, {info:info}, {note:note}, and {tip:tip}.

Remember that all these faces (and any user or package-defined ones) can
arbitrarily nest and overlap, {region,tip:like {bold,italic:so}}.")
```

The basic font-style attributes are **bold**, light, *italic*,
underline, and strikethrough.

In terms of color, we have named faces for the 16 standard terminal colors:

```
Since bright_black is effectively grey, we define two aliases for it:
grey and gray to allow for regional spelling differences.

To flip the foreground and background colors of some text, you can use the
inverse face, for example: some inverse text.

The intent-based basic faces are shadow (for dim but visible text),
region for selections, emphasis, and highlight.
As above, code is used for code-like text.

Lastly, we have the 'message severity' faces: error, warning,
success, info, note, and tip.

Remember that all these faces (and any user or package-defined ones) can
arbitrarily nest and overlap, like so.
```

102.6 API reference

Styling and Faces

`StyledStrings.StyledMarkup.@styled_str` – Macro.

```
@styled_str -> AnnotatedString
```

Construct a styled string. Within the string, {<specs>:<content>} structures apply the formatting to <content>, according to the list of comma-separated specifications <specs>. Each spec can either take the form of a face name, an inline face specification, or a key=value pair. The value must be wrapped by {...} should it contain any of the characters ,=:{}.

String interpolation with $ functions in the same way as regular strings, except quotes need to be escaped. Faces, keys, and values can also be interpolated with $.

Example

```
styled"The {bold:{italic:quick} {(foreground=#cd853f):brown} fox} jumped over the
↪  {link={https://en.wikipedia.org/wiki/Laziness}:lazy} dog"
```

Extended help

This macro can be described by the following EBNF grammar:

```
styledstring = { styled | interpolated | escaped | plain } ;

specialchar = '{' | '}' | '$' | '\"' ;
anychar = [\u0-\u1fffff] ;
plain = { anychar - specialchar } ;
escaped = '\\', specialchar ;

interpolated = '$', ? expr ? | '$(', ? expr ?, ')' ;

styled = '{', ws, annotations, ':', content, '}' ;
content = { interpolated | plain | escaped | styled } ;
```

```
annotations = annotation | annotations, ws, ',', ws, annotation ;
annotation = face | inlineface | keyvalue ;
ws = { ' ' | '\t' | '\n' } ; (* whitespace *)

face = facename | interpolated ;
facename = [A-Za-z0-9_]+ ;

inlineface = '(', ws, [ faceprop ], { ws, ',', faceprop }, ws, ')' ;
faceprop = [a-z]+, ws, '=', ws, ( [^,)]+ | interpolated) ;

keyvalue = key, ws, '=', ws, value ;
key = ( [^\0${}=,:], [^\0=,:]* ) | interpolated ;
value = simplevalue | curlybraced | interpolated ;
curlybraced = '{' { escaped | plain } '}' ;
simplevalue = [^${},:], [^,:]* ;
```

An extra stipulation not encoded in the above grammar is that `plain` should be a valid input to `unescape_string`, with `specialchar` kept.

The above grammar for `inlineface` is simplified, as the actual implementation is a bit more sophisticated. The full behaviour is given below.

```
faceprop = ( 'face', ws, '=', ws, ( ? string ? | interpolated ) ) |
           ( 'height', ws, '=', ws, ( ? number ? | interpolated ) ) |
           ( 'weight', ws, '=', ws, ( symbol | interpolated ) ) |
           ( 'slant', ws, '=', ws, ( symbol | interpolated ) ) |
           ( ( 'foreground' | 'fg' | 'background' | 'bg' ),
               ws, '=', ws, ( simplecolor | interpolated ) ) |
           ( 'underline', ws, '=', ws, ( underline | interpolated ) ) |
           ( 'strikethrough', ws, '=', ws, ( bool | interpolated ) ) |
           ( 'inverse', ws, '=', ws, ( bool | interpolated ) ) |
           ( 'inherit', ws, '=', ws, ( inherit | interpolated ) ) ;

nothing = 'nothing' ;
bool = 'true' | 'false' ;
symbol = [^ ,)]+ ;
hexcolor = ('#' | '0x'), [0-9a-f]{6} ;
simplecolor = hexcolor | symbol | nothing ;

underline = nothing | bool | simplecolor | underlinestyled;
underlinestyled = '(', ws, ('' | nothing | simplecolor | interpolated), ws,
                  ',', ws, ( symbol | interpolated ), ws ')' ;

inherit = ( '[', inheritval, { ',', inheritval }, ']' ) | inheritval;
inheritval = ws, ':'?, symbol ;
```

`StyledStrings.StyledMarkup.styled` – Function.

```
styled(content::AbstractString) -> AnnotatedString
```

Construct a styled string. Within the string, {<specs>:<content>} structures apply the formatting to <content>, according to the list of comma-separated specifications <specs>. Each spec can either take the form of a face name, an inline face specification, or a key=value pair. The value must be wrapped by `{...}` should it contain any of the characters `,=:{}`.

This is a functional equivalent of the `@styled_str` macro, just without interpolation capabilities.

`StyledStrings.Face` – Type.

A `Face` is a collection of graphical attributes for displaying text. Faces control how text is displayed in the terminal, and possibly other places too.

Most of the time, a `Face` will be stored in the global faces dicts as a unique association with a *face name* Symbol, and will be most often referred to by this name instead of the `Face` object itself.

Attributes

All attributes can be set via the keyword constructor, and default to `nothing`.

- `height` (an `Int` or `Float64`): The height in either deci-pt (when an `Int`), or as a factor of the base size (when a `Float64`).
- `weight` (a `Symbol`): One of the symbols (from faintest to densest) `:thin`, `:extralight`, `:light`, `:semilight`, `:normal`, `:medium`, `:semibold`, `:bold`, `:extrabold`, or `:black`. In terminals any weight greater than `:normal` is displayed as bold, and in terminals that support variable-brightness text, any weight less than `:normal` is displayed as faint.
- `slant` (a `Symbol`): One of the symbols `:italic`, `:oblique`, or `:normal`.
- `foreground` (a `SimpleColor`): The text foreground color.
- `background` (a `SimpleColor`): The text background color.
- `underline`, the text underline, which takes one of the following forms:
  - a `Bool`: Whether the text should be underlined or not.
  - a `SimpleColor`: The text should be underlined with this color.
  - a `Tuple{Nothing, Symbol}`: The text should be underlined using the style set by the Symbol, one of `:straight`, `:double`, `:curly`, `:dotted`, or `:dashed`.
  - a `Tuple{SimpleColor, Symbol}`: The text should be underlined in the specified SimpleColor, and using the style specified by the Symbol, as before.
- `strikethrough` (a `Bool`): Whether the text should be struck through.
- `inverse` (a `Bool`): Whether the foreground and background colors should be inverted.
- `inherit` (a `Vector{Symbol}`): Names of faces to inherit from, with earlier faces taking priority. All faces inherit from the `:default` face.

`StyledStrings.addface!` – Function.

```
addface!(name::Symbol => default::Face)
```

Create a new face by the name `name`. So long as no face already exists by this name, `default` is added to both `FACES.default` and (a copy of) to `FACES.current`, with the current value returned.

Should the face `name` already exist, `nothing` is returned.

Examples

```
julia> addface!(:mypkg_myface => Face(slant=:italic, underline=true))
Face (sample)
         slant: italic
     underline: true
```

StyledStrings.withfaces – Function.

```
withfaces(f, kv::Pair...)
withfaces(f, kvpair_itr)
```

Execute f with FACES.current temporarily modified by zero or more :name => val arguments kv, or kvpair_itr which produces kv-form values.

withfaces is generally used via the withfaces(kv...) do ... end syntax. A value of nothing can be used to temporarily unset a face (if it has been set). When withfaces returns, the original FACES.current has been restored.

Examples

```
julia> withfaces(:yellow => Face(foreground=:red), :green => :blue) do
           println(styled"{yellow:red} and {green:blue} mixed make {magenta:purple}")
       end
red and blue mixed make purple
```

StyledStrings.SimpleColor – Type.

```
struct SimpleColor
```

A basic representation of a color, intended for string styling purposes. It can either contain a named color (like :red), or an RGBTuple which is a NamedTuple specifying an r, g, b color with a bit-depth of 8.

Constructors

```
SimpleColor(name::Symbol)  # e.g. :red
SimpleColor(rgb::RGBTuple) # e.g. (r=1, b=2, g=3)
SimpleColor(r::Integer, b::Integer, b::Integer)
SimpleColor(rgb::UInt32)   # e.g. 0x123456
```

Also see tryparse(SimpleColor, rgb::String).

Base.parse – Method.

```
parse(::Type{SimpleColor}, rgb::String)
```

An analogue of tryparse(SimpleColor, rgb::String) (which see), that raises an error instead of returning nothing.

Base.tryparse – Method.

```
tryparse(::Type{SimpleColor}, rgb::String)
```

Attempt to parse `rgb` as a `SimpleColor`. If `rgb` starts with `#` and has a length of 7, it is converted into a `RGBTuple`-backed `SimpleColor`. If `rgb` starts with a-z, `rgb` is interpreted as a color name and converted to a `Symbol`-backed `SimpleColor`.

Otherwise, `nothing` is returned.

Examples

```
julia> tryparse(SimpleColor, "blue")
SimpleColor(blue)

julia> tryparse(SimpleColor, "#9558b2")
SimpleColor(#9558b2)

julia> tryparse(SimpleColor, "#nocolor")
```

`Base.merge` – Method.

```
merge(initial::StyledStrings.Face, others::StyledStrings.Face...)
```

Merge the properties of the `initial` face and `others`, with later faces taking priority.

This is used to combine the styles of multiple faces, and to resolve inheritance.

The `Tar` module provides a simple interface for handling tar archives, including creation of archives, extraction of selected files from an archive, and access to metadata.

Chapter 103

Tar

`Tar.create` – Function.

```
create(
    [ predicate, ] dir, [ tarball ];
    [ skeleton, ] [ portable = false ]
) -> tarball

    predicate :: String --> Bool
    dir       :: AbstractString
    tarball   :: Union{AbstractString, AbstractCmd, IO}
    skeleton  :: Union{AbstractString, AbstractCmd, IO}
    portable  :: Bool
```

Create a tar archive ("tarball") of the directory `dir`. The resulting archive is written to the path `tarball` or if no path is specified, a temporary path is created and returned by the function call. If `tarball` is an IO object then the tarball content is written to that handle instead (the handle is left open).

If a `predicate` function is passed, it is called on each system path that is encountered while recursively searching `dir` and path is only included in the tarball if `predicate(path)` is true. If `predicate(path)` returns false for a directory, then the directory is excluded entirely: nothing under that directory will be included in the archive.

If the `skeleton` keyword is passed then the file or IO handle given is used as a "skeleton" to generate the tarball. You create a skeleton file by passing the `skeleton` keyword to the `extract` command. If `create` is called with that skeleton file and the extracted files haven't changed, an identical tarball is recreated. The `skeleton` and `predicate` arguments cannot be used together.

If the `portable` flag is true then path names are checked for validity on Windows, which ensures that they don't contain illegal characters or have names that are reserved. See https://stackoverflow.com/a/31976060/659248 for details.

`Tar.extract` – Function.

```
extract(
    [ predicate, ] tarball, [ dir ];
    [ skeleton = <none>, ]
    [ copy_symlinks = <auto>, ]
    [ set_permissions = true, ]
```

```
) -> dir

    predicate       :: Header --> Bool
    tarball         :: Union{AbstractString, AbstractCmd, IO}
    dir             :: AbstractString
    skeleton        :: Union{AbstractString, AbstractCmd, IO}
    copy_symlinks   :: Bool
    set_permissions :: Bool
```

Extract a tar archive ("tarball") located at the path `tarball` into the directory `dir`. If `tarball` is an IO object instead of a path, then the archive contents will be read from that IO stream. The archive is extracted to `dir` which must either be an existing empty directory or a non-existent path which can be created as a new directory. If `dir` is not specified, the archive is extracted into a temporary directory which is returned by `extract`.

If a `predicate` function is passed, it is called on each `Header` object that is encountered while extracting `tarball` and the entry is only extracted if the `predicate(hdr)` is true. This can be used to selectively extract only parts of an archive, to skip entries that cause `extract` to throw an error, or to record what is extracted during the extraction process.

Before it is passed to the predicate function, the `Header` object is somewhat modified from the raw header in the tarball: the `path` field is normalized to remove `.` entries and replace multiple consecutive slashes with a single slash. If the entry has type `:hardlink`, the link target path is normalized the same way so that it will match the path of the target entry; the size field is set to the size of the target path (which must be an already-seen file).

If the `skeleton` keyword is passed then a "skeleton" of the extracted tarball is written to the file or IO handle given. This skeleton file can be used to recreate an identical tarball by passing the `skeleton` keyword to the `create` function. The `skeleton` and `predicate` arguments cannot be used together.

If `copy_symlinks` is `true` then instead of extracting symbolic links as such, they will be extracted as copies of what they link to if they are internal to the tarball and if it is possible to do so. Non-internal symlinks, such as a link to `/etc/passwd` will not be copied. Symlinks which are in any way cyclic will also not be copied and will instead be skipped. By default, `extract` will detect whether symlinks can be created in `dir` or not and will automatically copy symlinks if they cannot be created.

If `set_permissions` is `false`, no permissions are set on the extracted files.

`Tar.list` – Function.

```
list(tarball; [ strict = true ]) -> Vector{Header}
list(callback, tarball; [ strict = true ])

    callback  :: Header, [ <data> ] --> Any
    tarball   :: Union{AbstractString, AbstractCmd, IO}
    strict    :: Bool
```

List the contents of a tar archive ("tarball") located at the path `tarball`. If `tarball` is an IO handle, read the tar contents from that stream. Returns a vector of `Header` structs. See `Header` for details.

If a `callback` is provided then instead of returning a vector of headers, the callback is called on each `Header`. This can be useful if the number of items in the tarball is large or if you want examine items prior to an error in the tarball. If the `callback` function can accept a second argument of either type `Vector{UInt8}` or `Vector{Pair{Symbol, String}}` then it will be called with a representation of the

raw header data either as a single byte vector or as a vector of pairs mapping field names to the raw data for that field (if these fields are concatenated together, the result is the raw data of the header).

By default `list` will error if it encounters any tarball contents which the `extract` function would refuse to extract. With `strict=false` it will skip these checks and list all the the contents of the tar file whether `extract` would extract them or not. Beware that malicious tarballs can do all sorts of crafty and unexpected things to try to trick you into doing something bad.

If the `tarball` argument is a skeleton file (see `extract` and `create`) then `list` will detect that from the file header and appropriately list or iterate the headers of the skeleton file.

`Tar.rewrite` – Function.

```
rewrite(
    [ predicate, ] old_tarball, [ new_tarball ];
    [ portable = false, ]
) -> new_tarball

    predicate   :: Header --> Bool
    old_tarball :: Union{AbstractString, AbstractCmd, IO}
    new_tarball :: Union{AbstractString, AbstractCmd, IO}
    portable    :: Bool
```

Rewrite `old_tarball` to the standard format that `create` generates, while also checking that it doesn't contain anything that would cause `extract` to raise an error. This is functionally equivalent to doing

```
Tar.create(Tar.extract(predicate, old_tarball), new_tarball)
```

However, it never extracts anything to disk and instead uses the seek function to navigate the old tarball's data. If no `new_tarball` argument is passed, the new tarball is written to a temporary file whose path is returned.

If a `predicate` function is passed, it is called on each `Header` object that is encountered while extracting `old_tarball` and the entry is skipped unless `predicate(hdr)` is true. This can be used to selectively rewrite only parts of an archive, to skip entries that would cause `extract` to throw an error, or to record what content is encountered during the rewrite process.

Before it is passed to the predicate function, the `Header` object is somewhat modified from the raw header in the tarball: the `path` field is normalized to remove `.` entries and replace multiple consecutive slashes with a single slash. If the entry has type `:hardlink`, the link target path is normalized the same way so that it will match the path of the target entry; the size field is set to the size of the target path (which must be an already-seen file).

If the `portable` flag is true then path names are checked for validity on Windows, which ensures that they don't contain illegal characters or have names that are reserved. See https://stackoverflow.com/a/31976060/659248 for details.

`Tar.tree_hash` – Function.

```
tree_hash([ predicate, ] tarball;
          [ algorithm = "git-sha1", ]
          [ skip_empty = false ]) -> hash::String

    predicate  :: Header --> Bool
```

```
    tarball    :: Union{AbstractString, AbstractCmd, IO}
    algorithm  :: AbstractString
    skip_empty :: Bool
```

Compute a tree hash value for the file tree that the tarball contains. By default, this uses git's tree hashing algorithm with the SHA1 secure hash function (like current versions of git). This means that for any tarball whose file tree git can represent—i.e. one with only files, symlinks and non-empty directories—the hash value computed by this function will be the same as the hash value git would compute for that file tree. Note that tarballs can represent file trees with empty directories, which git cannot store, and this function can generate hashes for those, which will, by default (see `skip_empty` below for how to change this behavior), differ from the hash of a tarball which omits those empty directories. In short, the hash function agrees with git on all trees which git can represent, but extends (in a consistent way) the domain of hashable trees to other trees which git cannot represent.

If a `predicate` function is passed, it is called on each `Header` object that is encountered while processing `tarball` and an entry is only hashed if `predicate(hdr)` is true. This can be used to selectively hash only parts of an archive, to skip entries that cause `extract` to throw an error, or to record what is extracted during the hashing process.

Before it is passed to the predicate function, the `Header` object is somewhat modified from the raw header in the tarball: the `path` field is normalized to remove `.` entries and replace multiple consecutive slashes with a single slash. If the entry has type `:hardlink`, the link target path is normalized the same way so that it will match the path of the target entry; the size field is set to the size of the target path (which must be an already-seen file).

Currently supported values for `algorithm` are `git-sha1` (the default) and `git-sha256`, which uses the same basic algorithm as `git-sha1` but replaces the SHA1 hash function with SHA2-256, the hash function that git will transition to using in the future (due to known attacks on SHA1). Support for other file tree hashing algorithms may be added in the future.

The `skip_empty` option controls whether directories in the tarball which recursively contain no files or symlinks are included in the hash or ignored. In general, if you are hashing the content of a tarball or a file tree, you care about all directories, not just non-empty ones, so including these in the computed hash is the default. So why does this function even provide the option to skip empty directories? Because git refuses to store empty directories and will ignore them if you try to add them to a repo. So if you compute a reference tree hash by by adding files to a git repo and then asking git for the tree hash, the hash value that you get will match the hash value computed by `tree_hash` with `skip_empty=true`. In other words, this option allows `tree_hash` to emulate how git would hash a tree with empty directories. If you are hashing trees that may contain empty directories (i.e. do not come from a git repo), however, it is recommended that you hash them using a tool (such as this one) that does not ignore empty directories.

`Tar.Header` – Type.

The `Header` type is a struct representing the essential metadata for a single record in a tar file with this definition:

```
struct Header
    path :: String # path relative to the root
    type :: Symbol # type indicator (see below)
    mode :: UInt16 # mode/permissions (best viewed in octal)
    size :: Int64  # size of record data in bytes
    link :: String # target path of a symlink
```

```
end
```

Types are represented with the following symbols: `file`, `hardlink`, `symlink`, `chardev`, `blockdev`, `directory`, `fifo`, or for unknown types, the typeflag character as a symbol. Note that `extract` refuses to extract records types other than `file`, `symlink` and `directory`; `list` will only list other kinds of records if called with `strict=false`.

The tar format includes various other metadata about records, including user and group IDs, user and group names, and timestamps. The `Tar` package, by design, completely ignores these. When creating tar files, these fields are always set to zero/empty. When reading tar files, these fields are ignored aside from verifying header checksums for each header record for all fields.

Chapter 104

TOML

TOML.jl is a Julia standard library for parsing and writing TOML v1.0 files.

104.1 Parsing TOML data

```
julia> using TOML

julia> data = """
           [database]
           server = "192.168.1.1"
           ports = [ 8001, 8001, 8002 ]
       """;

julia> TOML.parse(data)
Dict{String, Any} with 1 entry:
  "database" => Dict{String, Any}("server"=>"192.168.1.1", "ports"=>[8001, 8001…
```

To parse a file, use `TOML.parsefile`. If the file has a syntax error, an exception is thrown:

```
julia> using TOML

julia> TOML.parse("""
           value = 0.0.0
       """)
ERROR: TOML Parser error:
none:1:16 error: failed to parse value
      value = 0.0.0
                ^
[...]
```

There are other versions of the parse functions (`TOML.tryparse` and `TOML.tryparsefile`) that instead of throwing exceptions on parser error returns a `TOML.ParserError` with information:

```
julia> using TOML

julia> err = TOML.tryparse("""
```

```
           value = 0.0.0
       """);

julia> err.type
ErrGenericValueError::ErrorType = 14

julia> err.line
1

julia> err.column
16
```

104.2 Exporting data to TOML file

The `TOML.print` function is used to print (or serialize) data into TOML format.

```
julia> using TOML

julia> data = Dict(
          "names" => ["Julia", "Julio"],
          "age" => [10, 20],
       );

julia> TOML.print(data)
names = ["Julia", "Julio"]
age = [10, 20]

julia> fname = tempname();

julia> open(fname, "w") do io
           TOML.print(io, data)
       end

julia> TOML.parsefile(fname)
Dict{String, Any} with 2 entries:
  "names" => ["Julia", "Julio"]
  "age"   => [10, 20]
```

Keys can be sorted according to some value

```
julia> using TOML

julia> TOML.print(Dict(
       "abc"  => 1,
       "ab"   => 2,
       "abcd" => 3,
       ); sorted=true, by=length)
ab = 2
abc = 1
abcd = 3
```

For custom structs, pass a function that converts the struct to a supported type

```
julia> using TOML

julia> struct MyStruct
           a::Int
           b::String
       end

julia> TOML.print(Dict("foo" => MyStruct(5, "bar"))) do x
           x isa MyStruct && return [x.a, x.b]
           error("unhandled type $(typeof(x))")
       end
foo = [5, "bar"]
```

104.3 References

TOML.parse – Function.

```
parse(x::Union{AbstractString, IO})
parse(p::Parser, x::Union{AbstractString, IO})
```

Parse the string or stream x, and return the resulting table (dictionary). Throw a ParserError upon failure.

See also TOML.tryparse.

TOML.parsefile – Function.

```
parsefile(f::AbstractString)
parsefile(p::Parser, f::AbstractString)
```

Parse file f and return the resulting table (dictionary). Throw a ParserError upon failure.

See also TOML.tryparsefile.

TOML.tryparse – Function.

```
tryparse(x::Union{AbstractString, IO})
tryparse(p::Parser, x::Union{AbstractString, IO})
```

Parse the string or stream x, and return the resulting table (dictionary). Return a ParserError upon failure.

See also TOML.parse.

TOML.tryparsefile – Function.

```
tryparsefile(f::AbstractString)
tryparsefile(p::Parser, f::AbstractString)
```

Parse file f and return the resulting table (dictionary). Return a ParserError upon failure.

See also TOML.parsefile.

TOML.print – Function.

```
print([to_toml::Function], io::IO [=stdout], data::AbstractDict; sorted=false, by=identity,
↪  inline_tables::IdSet{<:AbstractDict})
```

Write `data` as TOML syntax to the stream `io`. If the keyword argument `sorted` is set to `true`, sort tables according to the function given by the keyword argument `by`. If the keyword argument `inline_tables` is given, it should be a set of tables that should be printed "inline".

Julia 1.11

The `inline_tables` keyword argument is supported by Julia 1.11 or later.

The following data types are supported: `AbstractDict`, `AbstractVector`, `AbstractString`, `Integer`, `AbstractFloat`, `Bool`, `Dates.DateTime`, `Dates.Time`, `Dates.Date`. Note that the integers and floats need to be convertible to `Float64` and `Int64` respectively. For other data types, pass the function `to_toml` that takes the data types and returns a value of a supported type.

`TOML.Parser` – Type.

```
Parser()
```

Constructor for a TOML `Parser`. Note that in most cases one does not need to explicitly create a `Parser` but instead one directly uses `TOML.parsefile` or `TOML.parse`. Using an explicit parser will however reuse some internal data structures which can be beneficial for performance if a larger number of small files are parsed.

`TOML.ParserError` – Type.

```
ParserError
```

Type that is returned from `tryparse` and `tryparsefile` when parsing fails. It contains (among others) the following fields:

- `pos`, the position in the string when the error happened
- `table`, the result that so far was successfully parsed
- `type`, an error type, different for different types of errors

Chapter 105

Unicode

The Unicode module provides essential functionality for managing Unicode characters and strings. It includes validation, category determination, normalization, case transformation, and grapheme segmentation, enabling effective Unicode data handling.

Unicode – Module.

> The Unicode module provides essential functionality for managing Unicode characters and strings. It includes validation, category determination, normalization, case transformation, and grapheme segmentation, enabling effective Unicode data handling.

Unicode.julia_chartransform – Function.

```
Unicode.julia_chartransform(c::Union{Char,Integer})
```

> Map the Unicode character (Char) or codepoint (Integer) c to the corresponding "equivalent" character or codepoint, respectively, according to the custom equivalence used within the Julia parser (in addition to NFC normalization).
>
> For example, 'µ' (U+00B5 micro) is treated as equivalent to 'μ' (U+03BC mu) by Julia's parser, so julia_chartransform performs this transformation while leaving other characters unchanged:
>
> ```
> julia> Unicode.julia_chartransform('µ')
> 'µ': Unicode U+03BC (category Ll: Letter, lowercase)
>
> julia> Unicode.julia_chartransform('x')
> 'x': ASCII/Unicode U+0078 (category Ll: Letter, lowercase)
> ```
>
> julia_chartransform is mainly useful for passing to the Unicode.normalize function in order to mimic the normalization used by the Julia parser:
>
> ```
> julia> s = "µö"
> "µö"
>
> julia> s2 = Unicode.normalize(s, compose=true, stable=true,
> ↪  chartransform=Unicode.julia_chartransform)
> "µö"
>
> julia> collect(s2)
> ```

```
2-element Vector{Char}:
 'µ': Unicode U+03BC (category Ll: Letter, lowercase)
 'ö': Unicode U+00F6 (category Ll: Letter, lowercase)

julia> s2 == string(Meta.parse(s))
true
```

Julia 1.8

This function was introduced in Julia 1.8.

`Unicode.isassigned` – Function.

```
Unicode.isassigned(c)::Bool
```

Return `true` if the given char or integer is an assigned Unicode code point.

Examples

```
julia> Unicode.isassigned(101)
true

julia> Unicode.isassigned('\x01')
true
```

`Unicode.isequal_normalized` – Function.

```
isequal_normalized(s1::AbstractString, s2::AbstractString; casefold=false, stripmark=false,
↪  chartransform=identity)
```

Return whether `s1` and `s2` are canonically equivalent Unicode strings. If `casefold=true`, ignores case (performs Unicode case-folding); if `stripmark=true`, strips diacritical marks and other combining characters.

As with `Unicode.normalize`, you can also pass an arbitrary function via the `chartransform` keyword (mapping `Integer` codepoints to codepoints) to perform custom normalizations, such as `Unicode.julia_chartransform`.

Julia 1.8

The `isequal_normalized` function was added in Julia 1.8.

Examples

For example, the string `"noël"` can be constructed in two canonically equivalent ways in Unicode, depending on whether `"ë"` is formed from a single codepoint U+00EB or from the ASCII character `'e'` followed by the U+0308 combining-diaeresis character.

```
julia> s1 = "noël"
"noël"

julia> s2 = "noël"
"noël"
```

```
julia> s1 == s2
false

julia> isequal_normalized(s1, s2)
true

julia> isequal_normalized(s1, "noel", stripmark=true)
true

julia> isequal_normalized(s1, "NOËL", casefold=true)
true
```

`Unicode.normalize` – Function.

```
Unicode.normalize(s::AbstractString; keywords...)
Unicode.normalize(s::AbstractString, normalform::Symbol)
```

Normalize the string `s`. By default, canonical composition (`compose=true`) is performed without ensuring Unicode versioning stability (`compat=false`), which produces the shortest possible equivalent string but may introduce composition characters not present in earlier Unicode versions.

Alternatively, one of the four "normal forms" of the Unicode standard can be specified: `normalform` can be `:NFC`, `:NFD`, `:NFKC`, or `:NFKD`. Normal forms C (canonical composition) and D (canonical decomposition) convert different visually identical representations of the same abstract string into a single canonical form, with form C being more compact. Normal forms KC and KD additionally canonicalize "compatibility equivalents": they convert characters that are abstractly similar but visually distinct into a single canonical choice (e.g. they expand ligatures into the individual characters), with form KC being more compact.

Alternatively, finer control and additional transformations may be obtained by calling `Unicode.normalize(s; keywords...)`, where any number of the following boolean keywords options (which all default to `false` except for `compose`) are specified:

- `compose=false`: do not perform canonical composition
- `decompose=true`: do canonical decomposition instead of canonical composition (`compose=true` is ignored if present)
- `compat=true`: compatibility equivalents are canonicalized
- `casefold=true`: perform Unicode case folding, e.g. for case-insensitive string comparison
- `newline2lf=true`, `newline2ls=true`, or `newline2ps=true`: convert various newline sequences (LF, CRLF, CR, NEL) into a linefeed (LF), line-separation (LS), or paragraph-separation (PS) character, respectively
- `stripmark=true`: strip diacritical marks (e.g. accents)
- `stripignore=true`: strip Unicode's "default ignorable" characters (e.g. the soft hyphen or the left-to-right marker)
- `stripcc=true`: strip control characters; horizontal tabs and form feeds are converted to spaces; newlines are also converted to spaces unless a newline-conversion flag was specified
- `rejectna=true`: throw an error if unassigned code points are found
- `stable=true`: enforce Unicode versioning stability (never introduce characters missing from earlier Unicode versions)

You can also use the `chartransform` keyword (which defaults to `identity`) to pass an arbitrary *function* mapping `Integer` codepoints to codepoints, which is called on each character in `s` as it is processed, in order to perform arbitrary additional normalizations. For example, by passing `chartransform=Unicode.julia_chartransfor` you can apply a few Julia-specific character normalizations that are performed by Julia when parsing identifiers (in addition to NFC normalization: `compose=true, stable=true`).

For example, NFKC corresponds to the options `compose=true, compat=true, stable=true`.

Examples

```
julia> "é" == Unicode.normalize("é") #LHS: Unicode U+00e9, RHS: U+0065 & U+0301
true

julia> "µ" == Unicode.normalize("µ", compat=true) #LHS: Unicode U+03bc, RHS: Unicode U+00b5
true

julia> Unicode.normalize("JuLiA", casefold=true)
"julia"

julia> Unicode.normalize("JúLiA", stripmark=true)
"JuLiA"
```

Julia 1.8

The `chartransform` keyword argument requires Julia 1.8.

`Unicode.graphemes` – Function.

```
graphemes(s::AbstractString)::GraphemeIterator
```

Return an iterator over substrings of `s` that correspond to the extended graphemes in the string, as defined by Unicode UAX #29. (Roughly, these are what users would perceive as single characters, even though they may contain more than one codepoint; for example a letter combined with an accent mark is a single grapheme.)

```
graphemes(s::AbstractString, m:n)::SubString
```

Returns a `SubString` of `s` consisting of the `m`-th through `n`-th graphemes of the string `s`, where the second argument `m:n` is an integer-valued `AbstractUnitRange`.

Loosely speaking, this corresponds to the `m:n`-th user-perceived "characters" in the string. For example:

```
julia> s = graphemes("exposé", 3:6)
"posé"

julia> collect(s)
5-element Vector{Char}:
 'p': ASCII/Unicode U+0070 (category Ll: Letter, lowercase)
 'o': ASCII/Unicode U+006F (category Ll: Letter, lowercase)
 's': ASCII/Unicode U+0073 (category Ll: Letter, lowercase)
 'e': ASCII/Unicode U+0065 (category Ll: Letter, lowercase)
 '́': Unicode U+0301 (category Mn: Mark, nonspacing)
```

This consists of the 3rd to *7th* codepoints (Chars) in "exposé", because the grapheme "é" is actually *two* Unicode codepoints (an 'e' followed by an acute-accent combining character U+0301).

Because finding grapheme boundaries requires iteration over the string contents, the graphemes(s, m:n) function requires time proportional to the length of the string (number of codepoints) before the end of the substring.

Julia 1.9

The m:n argument of graphemes requires Julia 1.9.

Chapter 106

Unit Testing

106.1 Testing Base Julia

Julia is under rapid development and has an extensive test suite to verify functionality across multiple platforms. If you build Julia from source, you can run this test suite with `make test`. In a binary install, you can run the test suite using `Base.runtests()`.

`Base.runtests` – Function.

```
Base.runtests(tests=["all"]; ncores=ceil(Int, Sys.EFFECTIVE_CPU_THREADS / 2),
              exit_on_error=false, revise=false, propagate_project=true, [seed],
              ↪  [julia_args::Cmd])
```

Run the Julia unit tests listed in `tests`, which can be either a string or an array of strings, using `ncores` processors. If `exit_on_error` is `false`, when one test fails, all remaining tests in other files will still be run; they are otherwise discarded, when `exit_on_error == true`. If `revise` is `true`, the `Revise` package is used to load any modifications to `Base` or to the standard libraries before running the tests. If `propagate_project` is true the current project is propagated to the test environment. If a seed is provided via the keyword argument, it is used to seed the global RNG in the context where the tests are run; otherwise the seed is chosen randomly. The argument `julia_args` can be used to pass custom `julia` command line flags to the test process.

source

106.2 Basic Unit Tests

The `Test` module provides simple *unit testing* functionality. Unit testing is a way to see if your code is correct by checking that the results are what you expect. It can be helpful to ensure your code still works after you make changes, and can be used when developing as a way of specifying the behaviors your code should have when complete. You may also want to look at the documentation for adding tests to your Julia Package.

Simple unit testing can be performed with the `@test` and `@test_throws` macros:

`Test.@test` – Macro.

```
@test ex
@test f(args...) key=val ...
@test ex broken=true
```

```
@test ex skip=true
```

Test that the expression ex evaluates to `true`. If executed inside a `@testset`, return a `Pass` `Result` if it does, a `Fail` `Result` if it is `false`, and an `Error` `Result` if it could not be evaluated. If executed outside a `@testset`, throw an exception instead of returning `Fail` or `Error`.

Examples

```
julia> @test true
Test Passed

julia> @test [1, 2] + [2, 1] == [3, 3]
Test Passed
```

The `@test f(args...) key=val...` form is equivalent to writing `@test f(args..., key=val...)` which can be useful when the expression is a call using infix syntax such as approximate comparisons:

```
julia> @test π ≈ 3.14 atol=0.01
Test Passed
```

This is equivalent to the uglier test `@test ≈(π, 3.14, atol=0.01)`. It is an error to supply more than one expression unless the first is a call expression and the rest are assignments (`k=v`).

You can use any key for the `key=val` arguments, except for `broken` and `skip`, which have special meanings in the context of `@test`:

- `broken=cond` indicates a test that should pass but currently consistently fails when `cond==true`. Tests that the expression ex evaluates to `false` or causes an exception. Returns a `Broken` `Result` if it does, or an `Error` `Result` if the expression evaluates to `true`. Regular `@test ex` is evaluated when `cond==false`.
- `skip=cond` marks a test that should not be executed but should be included in test summary reporting as `Broken`, when `cond==true`. This can be useful for tests that intermittently fail, or tests of not-yet-implemented functionality. Regular `@test ex` is evaluated when `cond==false`.

Examples

```
julia> @test 2 + 2 ≈ 6 atol=1 broken=true
Test Broken
  Expression: ≈(2 + 2, 6, atol = 1)

julia> @test 2 + 2 ≈ 5 atol=1 broken=false
Test Passed

julia> @test 2 + 2 == 5 skip=true
Test Broken
  Skipped: 2 + 2 == 5

julia> @test 2 + 2 == 4 skip=false
Test Passed
```

Julia 1.7

The `broken` and `skip` keyword arguments require at least Julia 1.7.

`Test.@test_throws` – Macro.

```
@test_throws exception expr
@test_throws extype pattern expr
```

Tests that the expression `expr` throws `exception`. The exception may specify either a type, a string, regular expression, or list of strings occurring in the displayed error message, a matching function, or a value (which will be tested for equality by comparing fields).

In the two-argument form, `@test_throws exception expr`, the `exception` can be a type or a pattern.

In the three-argument form, `@test_throws extype pattern expr`, both the exception type and a message pattern are tested. The `extype` must be a type, and `pattern` may be a string, regular expression, or list of strings occurring in the displayed error message, a matching function, or a value.

Note that `@test_throws` does not support a trailing keyword form.

Julia 1.8

The ability to specify anything other than a type or a value as `exception` requires Julia v1.8 or later.

Julia 1.13

The three-argument form `@test_throws extype pattern expr` requires Julia v1.12 or later.

Examples

```
julia> @test_throws BoundsError [1, 2, 3][4]
Test Passed
      Thrown: BoundsError

julia> @test_throws DimensionMismatch [1, 2, 3] + [1, 2]
Test Passed
      Thrown: DimensionMismatch

julia> @test_throws "Try sqrt(Complex" sqrt(-1)
Test Passed
     Message: "DomainError with -1.0:\nsqrt was called with a negative real argument but will
     ↪  only return a complex result if called with a complex argument. Try
     ↪  sqrt(Complex(x))."

julia> @test_throws ErrorException "error foo" error("error foo 1")
Test Passed
      Thrown: ErrorException
```

In the third example, instead of matching a single string it could alternatively have been performed with:

- `["Try", "Complex"]` (a list of strings)

- `r"Try sqrt\([Cc]omplex"` (a regular expression)
- `str -> occursin("complex", str)` (a matching function)

In the final example, both the exception type (`ErrorException`) and message pattern (`"error foo"`) are tested.

For example, suppose we want to check our new function `foo(x)` works as expected:

```
julia> using Test

julia> foo(x) = length(x)^2
foo (generic function with 1 method)
```

If the condition is true, a `Pass` is returned:

```
julia> @test foo("bar") == 9
Test Passed

julia> @test foo("fizz") >= 10
Test Passed
```

If the condition is false, then a `Fail` is returned and an exception is thrown:

```
julia> @test foo("f") == 20
Test Failed at none:1
  Expression: foo("f") == 20
   Evaluated: 1 == 20
ERROR: There was an error during testing
```

If the condition could not be evaluated because an exception was thrown, which occurs in this case because `length` is not defined for symbols, an `Error` object is returned and an exception is thrown:

```
julia> @test foo(:cat) == 1
Error During Test
  Test threw an exception of type MethodError
  Expression: foo(:cat) == 1
  MethodError: no method matching length(::Symbol)
  The function `length` exists, but no method is defined for this combination of argument types.

  Closest candidates are:
    length(::SimpleVector) at essentials.jl:256
    length(::Base.MethodList) at reflection.jl:521
    length(::MethodTable) at reflection.jl:597
    ...
  Stacktrace:
  [...]
ERROR: There was an error during testing
```

If we expect that evaluating an expression *should* throw an exception, then we can use `@test_throws` to check that this occurs:

```
julia> @test_throws MethodError foo(:cat)
Test Passed
      Thrown: MethodError
```

106.3 Working with Test Sets

Typically a large number of tests are used to make sure functions work correctly over a range of inputs. In the event a test fails, the default behavior is to throw an exception immediately. However, it is normally preferable to run the rest of the tests first to get a better picture of how many errors there are in the code being tested.

Note

The `@testset` will create a local scope of its own when running the tests in it.

The `@testset` macro can be used to group tests into *sets*. All the tests in a test set will be run, and at the end of the test set a summary will be printed. If any of the tests failed, or could not be evaluated due to an error, the test set will then throw a `TestSetException`.

`Test.@testset` – Macro.

```
@testset [CustomTestSet] [options...] ["description"] begin test_ex end
@testset [CustomTestSet] [options...] ["description $v"] for v in itr test_ex end
@testset [CustomTestSet] [options...] ["description $v, $w"] for v in itrv, w in itrw test_ex
↪  end
@testset [CustomTestSet] [options...] ["description"] test_func()
@testset let v = v, w = w; test_ex; end
```

With begin/end or function call

When @testset is used, with begin/end or a single function call, the macro starts a new test set in which to evaluate the given expression.

If no custom testset type is given it defaults to creating a `DefaultTestSet`. `DefaultTestSet` records all the results and, if there are any `Fails` or `Errors`, throws an exception at the end of the top-level (non-nested) test set, along with a summary of the test results.

Any custom testset type (subtype of `AbstractTestSet`) can be given and it will also be used for any nested `@testset` invocations. The given options are only applied to the test set where they are given. The default test set type accepts the following options:

- `verbose::Bool`: if `true`, the result summary of the nested testsets is shown even when they all pass (the default is `false`).
- `showtiming::Bool`: if `true`, the duration of each displayed testset is shown (the default is `true`).
- `failfast::Bool`: if `true`, any test failure or error will cause the testset and any child testsets to return immediately (the default is `false`). This can also be set globally via the env var `JULIA_TEST_FAILFAST`.
- `rng::Random.AbstractRNG`: use the given random number generator (RNG) as the global one for the testset. `rng` must be `copy!`-able. This option can be useful to locally reproduce stochastic test failures which only depend on the state of the global RNG.

Julia 1.8

`@testset test_func()` requires at least Julia 1.8.

Julia 1.9

`failfast` requires at least Julia 1.9.

Julia 1.12

The `rng` option requires at least Julia 1.12.

The description string accepts interpolation from the loop indices. If no description is provided, one is constructed based on the variables. If a function call is provided, its name will be used. Explicit description strings override this behavior.

By default the `@testset` macro will return the testset object itself, though this behavior can be customized in other testset types. If a `for` loop is used then the macro collects and returns a list of the return values of the `finish` method, which by default will return a list of the testset objects used in each iteration.

Before the execution of the body of a `@testset`, there is an implicit call to `copy!(Random.default_rng(), rng)` where `rng` is the RNG of the current task, or the value of the RNG passed via the `rng` option. Moreover, after the execution of the body, the state of the global RNG is restored to what it was before the `@testset`. This is meant to ease reproducibility in case of failure, and to allow seamless re-arrangements of `@testsets` regardless of their side-effect on the global RNG state.

RNG of nested testsets

Unless changed with the `rng` option, the same RNG is set at the beginning of all nested testsets. The RNG printed to screen when a testset has failures is the global RNG of the outermost testset even if inner testsets have different RNGs manually set by the user.

Examples

```
julia> @testset "trigonometric identities" begin
           θ = 2/3*π
           @test sin(-θ) ≈ -sin(θ)
           @test cos(-θ) ≈ cos(θ)
           @test sin(2θ) ≈ 2*sin(θ)*cos(θ)
           @test cos(2θ) ≈ cos(θ)^2 - sin(θ)^2
       end;
Test Summary:            | Pass  Total  Time
trigonometric identities |    4      4  0.2s
```

`@testset for`

When `@testset for` is used, the macro starts a new test set for each iteration of the provided loop. The semantics of each test set are otherwise identical to that of the `begin`/`end` case (as if used for each loop iteration).

`@testset let`

When `@testset let` is used, the macro starts a *transparent* test set with the given object added as a context object to any failing or erroring test contained therein. This is useful when performing a set of related tests on one larger object and it is desirable to print this larger object when any of the individual tests fail. Transparent test sets do not introduce additional levels of nesting in the test set hierarchy and are passed through directly to the parent test set (with the context object appended to any failing tests.)

Julia 1.9

`@testset let` requires at least Julia 1.9.

Julia 1.10

Multiple `let` assignments are supported since Julia 1.10.

Julia 1.13

Context is shown when a test errors since Julia 1.13.

Special implicit world age increment for `@testset begin` and `@testset for`

World age inside `@testset begin` and inside the loop body of `@testset for` increments implicitly after every statement. This matches the behavior of ordinary toplevel code, but not that of ordinary `begin`/`end` blocks or `for` loops, i.e. with respect to world age, `@testset begin` and `@testset for` behave as if their bodies were written at toplevel.

Examples

```
julia> @testset let logi = log(im)
           @test imag(logi) == π/2
           @test !iszero(real(logi))
       end
Test Failed at none:3
  Expression: !(iszero(real(logi)))
   Evaluated: !(iszero(0.0))
     Context: logi = 0.0 + 1.5707963267948966im
ERROR: There was an error during testing

julia> @testset let logi = log(im), op = !iszero
           @test imag(logi) == π/2
           @test op(real(logi))
       end
Test Failed at none:3
  Expression: op(real(logi))
   Evaluated: op(0.0)
     Context: logi = 0.0 + 1.5707963267948966im
              op = !iszero
ERROR: There was an error during testing
```

`Test.TestSetException` – Type.

```
TestSetException
```

Thrown when a test set finishes and not all tests passed.

We can put our tests for the `foo(x)` function in a test set:

```
julia> @testset "Foo Tests" begin
           @test foo("a")   == 1
           @test foo("ab")  == 4
           @test foo("abc") == 9
       end;
Test Summary: | Pass  Total  Time
Foo Tests     |    3      3  0.0s
```

Test sets can also be nested:

```
julia> @testset "Foo Tests" begin
           @testset "Animals" begin
               @test foo("cat") == 9
               @test foo("dog") == foo("cat")
           end
           @testset "Arrays $i" for i in 1:3
               @test foo(zeros(i)) == i^2
               @test foo(fill(1.0, i)) == i^2
           end
       end;
Test Summary: | Pass  Total  Time
Foo Tests     |    8      8  0.0s
```

As well as call functions:

```
julia> f(x) = @test isone(x)
f (generic function with 1 method)

julia> @testset f(1);
Test Summary: | Pass  Total  Time
f             |    1      1  0.0s
```

This can be used to allow for factorization of test sets, making it easier to run individual test sets by running the associated functions instead. Note that in the case of functions, the test set will be given the name of the called function. In the event that a nested test set has no failures, as happened here, it will be hidden in the summary, unless the `verbose=true` option is passed:

```
julia> @testset verbose = true "Foo Tests" begin
           @testset "Animals" begin
               @test foo("cat") == 9
               @test foo("dog") == foo("cat")
           end
           @testset "Arrays $i" for i in 1:3
               @test foo(zeros(i)) == i^2
               @test foo(fill(1.0, i)) == i^2
           end
```

```
      end;
Test Summary: | Pass  Total  Time
Foo Tests     |    8      8  0.0s
  Animals     |    2      2  0.0s
  Arrays 1    |    2      2  0.0s
  Arrays 2    |    2      2  0.0s
  Arrays 3    |    2      2  0.0s
```

Environment Variable Support

The `Test` module supports the `JULIA_TEST_VERBOSE` environment variable for controlling verbose behavior globally:

- When `JULIA_TEST_VERBOSE=true`, testsets will automatically use `verbose=true` by default, and additionally print "Starting testset" and "Finished testset" messages with timing information as testsets are entered and exited.
- When `JULIA_TEST_VERBOSE=false` or unset, testsets use `verbose=false` by default and no entry/exit messages are printed.

This environment variable provides a convenient way to enable comprehensive verbose output for debugging test suites without modifying the test code itself.

```
$ JULIA_TEST_VERBOSE=true julia -e '
using Test
@testset "Example" begin
    @test 1 + 1 == 2
    @testset "Nested" begin
        @test 2 * 2 == 4
    end
end'

Starting testset: Example
  Starting testset: Nested
  Finished testset: Nested (0.0s)
Finished testset: Example (0.0s)
Test Summary: | Pass  Total  Time
Example       |    2      2  0.0s
  Nested      |    1      1  0.0s
```

If we do have a test failure, only the details for the failed test sets will be shown:

```
julia> @testset "Foo Tests" begin
           @testset "Animals" begin
               @testset "Felines" begin
                   @test foo("cat") == 9
               end
               @testset "Canines" begin
                   @test foo("dog") == 9
               end
           end
```

```
          @testset "Arrays" begin
              @test foo(zeros(2)) == 4
              @test foo(fill(1.0, 4)) == 15
          end
      end

Arrays: Test Failed
  Expression: foo(fill(1.0, 4)) == 15
   Evaluated: 16 == 15
[...]
Test Summary: | Pass  Fail  Total  Time
Foo Tests     |    3     1      4  0.0s
  Animals     |    2            2  0.0s
  Arrays      |    1     1      2  0.0s
ERROR: Some tests did not pass: 3 passed, 1 failed, 0 errored, 0 broken.
```

106.4 Testing Log Statements

One can use the `@test_logs` macro to test log statements, or use a `TestLogger`.

`Test.@test_logs` – Macro.

```
@test_logs [log_patterns...] [keywords] expression
```

Collect a list of log records generated by `expression` using `collect_test_logs`, check that they match the sequence `log_patterns`, and return the value of `expression`. The `keywords` provide some simple filtering of log records: the `min_level` keyword controls the minimum log level which will be collected for the test, the `match_mode` keyword defines how matching will be performed (the default `:all` checks that all logs and patterns match pairwise; use `:any` to check that the pattern matches at least once somewhere in the sequence.)

The most useful log pattern is a simple tuple of the form `(level,message)`. A different number of tuple elements may be used to match other log metadata, corresponding to the arguments to passed to `AbstractLogger` via the `handle_message` function: `(level,message,module,group,id,file,line)`. Elements which are present will be matched pairwise with the log record fields using `==` by default, with the special cases that `Symbol`s may be used for the standard log levels, and Regexs in the pattern will match string or Symbol fields using `occursin`.

Examples

Consider a function which logs a warning, and several debug messages:

```
function foo(n)
    @info "Doing foo with n=$n"
    for i=1:n
        @debug "Iteration $i"
    end
    42
end
```

We can test the info message using

```
@test_logs (:info,"Doing foo with n=2") foo(2)
```

If we also wanted to test the debug messages, these need to be enabled with the `min_level` keyword:

```
using Logging
@test_logs (:info,"Doing foo with n=2") (:debug,"Iteration 1") (:debug,"Iteration 2")
↪  min_level=Logging.Debug foo(2)
```

If you want to test that some particular messages are generated while ignoring the rest, you can set the keyword `match_mode=:any`:

```
using Logging
@test_logs (:info,) (:debug,"Iteration 42") min_level=Logging.Debug match_mode=:any foo(100)
```

The macro may be chained with `@test` to also test the returned value:

```
@test (@test_logs (:info,"Doing foo with n=2") foo(2)) == 42
```

If you want to test for the absence of warnings, you can omit specifying log patterns and set the `min_level` accordingly:

```
# test that the expression logs no messages when the logger level is warn:
@test_logs min_level=Logging.Warn @info("Some information") # passes
@test_logs min_level=Logging.Warn @warn("Some information") # fails
```

If you want to test the absence of warnings (or error messages) in `stderr` which are not generated by `@warn`, see `@test_nowarn`.

`Test.TestLogger` – Type.

```
TestLogger(; min_level=Info, catch_exceptions=false)
```

Create a `TestLogger` which captures logged messages in its `logs::Vector{LogRecord}` field.

Set `min_level` to control the `LogLevel`, `catch_exceptions` for whether or not exceptions thrown as part of log event generation should be caught, and `respect_maxlog` for whether or not to follow the convention of logging messages with `maxlog=n` for some integer `n` at most `n` times.

See also: `LogRecord`.

Examples

```
julia> using Test, Logging

julia> f() = @info "Hi" number=5;

julia> test_logger = TestLogger();

julia> with_logger(test_logger) do
           f()
           @info "Bye!"
       end

julia> @test test_logger.logs[1].message == "Hi"
Test Passed
```

```
julia> @test test_logger.logs[1].kwargs[:number] == 5
Test Passed

julia> @test test_logger.logs[2].message == "Bye!"
Test Passed
```

`Test.LogRecord` – Type.

```
LogRecord
```

Stores the results of a single log event. Fields:

- `level`: the `LogLevel` of the log message
- `message`: the textual content of the log message
- `_module`: the module of the log event
- `group`: the logging group (by default, the name of the file containing the log event)
- `id`: the ID of the log event
- `file`: the file containing the log event
- `line`: the line within the file of the log event
- `kwargs`: any keyword arguments passed to the log event

106.5 Other Test Macros

As calculations on floating-point values can be imprecise, you can perform approximate equality checks using either `@test a ≈ b` (where ≈, typed via tab completion of `\approx`, is the `isapprox` function) or use `isapprox` directly.

```
julia> @test 1 ≈ 0.999999999
Test Passed

julia> @test 1 ≈ 0.999999
Test Failed at none:1
  Expression: 1 ≈ 0.999999
ERROR: There was an error during testing
```

You can specify relative and absolute tolerances by setting the `rtol` and `atol` keyword arguments of `isapprox`, respectively, after the ≈ comparison:

```
julia> @test 1 ≈ 0.999999  rtol=1e-5
Test Passed
```

Note that this is not a specific feature of the ≈ but rather a general feature of the `@test` macro: `@test a <op> b key=val` is transformed by the macro into `@test op(a, b, key=val)`. It is, however, particularly useful for ≈ tests.

`Test.@inferred` – Macro.

```
@inferred [AllowedType] f(x)
```

Tests that the call expression `f(x)` returns a value of the same type inferred by the compiler. It is useful to check for type stability.

`f(x)` can be any call expression. Returns the result of `f(x)` if the types match, and an `Error Result` if it finds different types.

Optionally, `AllowedType` relaxes the test, by making it pass when either the type of `f(x)` matches the inferred type modulo `AllowedType`, or when the return type is a subtype of `AllowedType`. This is useful when testing type stability of functions returning a small union such as `Union{Nothing, T}` or `Union{Missing, T}`.

```
julia> f(a) = a > 1 ? 1 : 1.0
f (generic function with 1 method)

julia> typeof(f(2))
Int64

julia> @code_warntype f(2)
MethodInstance for f(::Int64)
  from f(a) @ Main none:1
Arguments
  #self#::Core.Const(f)
  a::Int64
Body::UNION{FLOAT64, INT64}
1 — %1 = :>::Core.Const(>)
│   %2 = (%1)(a, 1)::Bool
└──      goto #3 if not %2
2 —      return 1
3 —      return 1.0

julia> @inferred f(2)
ERROR: return type Int64 does not match inferred return type Union{Float64, Int64}
[...]

julia> @inferred max(1, 2)
2

julia> g(a) = a < 10 ? missing : 1.0
g (generic function with 1 method)

julia> @inferred g(20)
ERROR: return type Float64 does not match inferred return type Union{Missing, Float64}
[...]

julia> @inferred Missing g(20)
1.0

julia> h(a) = a < 10 ? missing : f(a)
h (generic function with 1 method)

julia> @inferred Missing h(20)
ERROR: return type Int64 does not match inferred return type Union{Missing, Float64, Int64}
[...]
```

`Test.@test_deprecated` – Macro.

```
@test_deprecated [pattern] expression
```

When `--depwarn=yes`, test that `expression` emits a deprecation warning and return the value of `expression`. The log message string will be matched against `pattern` which defaults to `r"deprecated"i`.

When `--depwarn=no`, simply return the result of executing `expression`. When `--depwarn=error`, check that an ErrorException is thrown.

Examples

```
# Deprecated in julia 0.7
@test_deprecated num2hex(1)

# The returned value can be tested by chaining with @test:
@test (@test_deprecated num2hex(1)) == "0000000000000001"
```

`Test.@test_warn` – Macro.

```
@test_warn msg expr
```

Test whether evaluating `expr` results in `stderr` output that contains the `msg` string or matches the `msg` regular expression. If `msg` is a boolean function, tests whether `msg(output)` returns `true`. If `msg` is a tuple or array, checks that the error output contains/matches each item in `msg`. Returns the result of evaluating `expr`.

See also `@test_nowarn` to check for the absence of error output.

Note: Warnings generated by `@warn` cannot be tested with this macro. Use `@test_logs` instead.

`Test.@test_nowarn` – Macro.

```
@test_nowarn expr
```

Test whether evaluating `expr` results in empty `stderr` output (no warnings or other messages). Returns the result of evaluating `expr`.

Note: The absence of warnings generated by `@warn` cannot be tested with this macro. Use `@test_logs` instead.

106.6 Broken Tests

If a test fails consistently it can be changed to use the `@test_broken` macro. This will denote the test as `Broken` if the test continues to fail and alerts the user via an `Error` if the test succeeds.

`Test.@test_broken` – Macro.

```
@test_broken ex
@test_broken f(args...) key=val ...
```

Indicates a test that should pass but currently consistently fails. Tests that the expression `ex` evaluates to `false` or causes an exception. Returns a `Broken Result` if it does, or an `Error Result` if the expression evaluates to `true`. This is equivalent to `@test ex broken=true`.

The `@test_broken f(args...) key=val...` form works as for the `@test` macro.

Examples

```
julia> @test_broken 1 == 2
Test Broken
  Expression: 1 == 2

julia> @test_broken 1 == 2 atol=0.1
Test Broken
  Expression: ==(1, 2, atol = 0.1)
```

`@test_skip` is also available to skip a test without evaluation, but counting the skipped test in the test set reporting. The test will not run but gives a `Broken Result`.

`Test.@test_skip` – Macro.

```
@test_skip ex
@test_skip f(args...) key=val ...
```

Marks a test that should not be executed but should be included in test summary reporting as `Broken`. This can be useful for tests that intermittently fail, or tests of not-yet-implemented functionality. This is equivalent to `@test ex skip=true`.

The `@test_skip f(args...) key=val...` form works as for the `@test` macro.

Examples

```
julia> @test_skip 1 == 2
Test Broken
  Skipped: 1 == 2

julia> @test_skip 1 == 2 atol=0.1
Test Broken
  Skipped: ==(1, 2, atol = 0.1)
```

106.7 Test result types

`Test.Result` – Type.

```
Test.Result
```

All tests produce a result object. This object may or may not be stored, depending on whether the test is part of a test set.

`Test.Pass` – Type.

```
Test.Pass <: Test.Result
```

The test condition was true, i.e. the expression evaluated to true or the correct exception was thrown.

`Test.Fail` – Type.

```
Test.Fail <: Test.Result
```

The test condition was false, i.e. the expression evaluated to false or the correct exception was not thrown.

`Test.Error` – Type.

```
Test.Error <: Test.Result
```

The test condition couldn't be evaluated due to an exception, or it evaluated to something other than a `Bool`. In the case of `@test_broken` it is used to indicate that an unexpected `Pass` `Result` occurred.

`Test.Broken` – Type.

```
Test.Broken <: Test.Result
```

The test condition is the expected (failed) result of a broken test, or was explicitly skipped with `@test_skip`.

106.8 Creating Custom `AbstractTestSet` Types

Packages can create their own `AbstractTestSet` subtypes by implementing the `record` and `finish` methods. The subtype should have a one-argument constructor taking a description string, with any options passed in as keyword arguments.

`Test.record` – Function.

```
record(ts::AbstractTestSet, res::Result)
```

Record a result to a testset. This function is called by the `@testset` infrastructure each time a contained `@test` macro completes, and is given the test result (which could be an `Error`). This will also be called with an `Error` if an exception is thrown inside the test block but outside of a `@test` context.

`Test.finish` – Function.

```
finish(ts::AbstractTestSet)
```

Do any final processing necessary for the given testset. This is called by the `@testset` infrastructure after a test block executes.

Custom `AbstractTestSet` subtypes should call `record` on their parent (if there is one) to add themselves to the tree of test results. This might be implemented as:

```
if get_testset_depth() != 0
    # Attach this test set to the parent test set
    parent_ts = get_testset()
    record(parent_ts, self)
    return self
end
```

`Test` takes responsibility for maintaining a stack of nested testsets as they are executed, but any result accumulation is the responsibility of the `AbstractTestSet` subtype. You can access this stack with the `get_testset` and `get_testset_depth` methods. Note that these functions are not exported.

`Test.get_testset` – Function.

```
get_testset()
```

Retrieve the active test set from the task's local storage. If no test set is active, use the fallback default test set.

`Test.get_testset_depth` – Function.

```
get_testset_depth()
```

Return the number of active test sets, not including the default test set

`Test` also makes sure that nested `@testset` invocations use the same `AbstractTestSet` subtype as their parent unless it is set explicitly. It does not propagate any properties of the testset. Option inheritance behavior can be implemented by packages using the stack infrastructure that `Test` provides.

Defining a basic `AbstractTestSet` subtype might look like:

```
import Test: Test, record, finish
using Test: AbstractTestSet, Result, Pass, Fail, Error
using Test: get_testset_depth, get_testset
struct CustomTestSet <: Test.AbstractTestSet
    description::AbstractString
    foo::Int
    results::Vector
    # constructor takes a description string and options keyword arguments
    CustomTestSet(desc; foo=1) = new(desc, foo, [])
end

record(ts::CustomTestSet, child::AbstractTestSet) = push!(ts.results, child)
record(ts::CustomTestSet, res::Result) = push!(ts.results, res)
function finish(ts::CustomTestSet)
    # just record if we're not the top-level parent
    if get_testset_depth() > 0
        record(get_testset(), ts)
        return ts
    end

    # so the results are printed if we are at the top level
    Test.print_test_results(ts)
    return ts
end
```

And using that testset looks like:

```
@testset CustomTestSet foo=4 "custom testset inner 2" begin
    # this testset should inherit the type, but not the argument.
    @testset "custom testset inner" begin
        @test true
    end
end
```

In order to use a custom testset and have the recorded results printed as part of any outer default testset, also define `Test.get_test_counts`. This might look like so:

```
using Test: AbstractTestSet, Pass, Fail, Error, Broken, get_test_counts, TestCounts,
↪  format_duration

function Test.get_test_counts(ts::CustomTestSet)
```

```
    passes, fails, errors, broken = 0, 0, 0, 0
    # cumulative results
    c_passes, c_fails, c_errors, c_broken = 0, 0, 0, 0

    for t in ts.results
        # count up results
        isa(t, Pass)   && (passes += 1)
        isa(t, Fail)   && (fails  += 1)
        isa(t, Error)  && (errors += 1)
        isa(t, Broken) && (broken += 1)
        # handle children
        if isa(t, AbstractTestSet)
            tc = get_test_counts(t)::TestCounts
            c_passes += tc.passes + tc.cumulative_passes
            c_fails  += tc.fails + tc.cumulative_fails
            c_errors += tc.errors + tc.cumulative_errors
            c_broken += tc.broken + tc.cumulative_broken
        end
    end
    # get a duration, if we have one
    duration = format_duration(ts)
    return TestCounts(true, passes, fails, errors, broken, c_passes, c_fails, c_errors, c_broken,
    ↪  duration)
end
```

`Test.TestCounts` – Type.

```
TestCounts
```

Holds the state for recursively gathering the results of a test set for display purposes.

Fields:

- `customized`: Whether the function `get_test_counts` was customized for the `AbstractTestSet` this counts object is for. If a custom method was defined, always pass `true` to the constructor.
- `passes`: The number of passing `@test` invocations.
- `fails`: The number of failing `@test` invocations.
- `errors`: The number of erroring `@test` invocations.
- `broken`: The number of broken `@test` invocations.
- `passes`: The cumulative number of passing `@test` invocations.
- `fails`: The cumulative number of failing `@test` invocations.
- `errors`: The cumulative number of erroring `@test` invocations.
- `broken`: The cumulative number of broken `@test` invocations.
- `duration`: The total duration the `AbstractTestSet` in question ran for, as a formatted `String`.

`Test.get_test_counts` – Function.

" get*test*counts(::AbstractTestSet)::TestCounts

Recursive function that counts the number of test results of each type directly in the testset, and totals across the child testsets.

Custom `AbstractTestSet` should implement this function to get their totals counted & displayed with `DefaultTestSet` as well.

If this is not implemented for a custom `TestSet`, the printing falls back to reporting `x` for failures and `?s` for the duration.

`Test.format_duration` – Function.

```
format_duration(::AbstractTestSet)
```

Return a formatted string for printing the duration the testset ran for.

If not defined, falls back to `"?s"`.

`Test.print_test_results` – Function.

```
print_test_results([io::IO], ts::AbstractTestSet, depth_pad=0)
```

Print the results of an `AbstractTestSet` as a formatted table.

depth_pad refers to how much padding should be added in front of all output. If `io` is not provided, defaults to `stdout`.

Called inside of `Test.finish`, if the `finish`ed testset is the topmost testset.

106.9 Test utilities

`Test.GenericArray` – Type.

The `GenericArray` can be used to test generic array APIs that program to the `AbstractArray` interface, in order to ensure that functions can work with array types besides the standard `Array` type.

`Test.GenericDict` – Type.

The `GenericDict` can be used to test generic dict APIs that program to the `AbstractDict` interface, in order to ensure that functions can work with associative types besides the standard `Dict` type.

`Test.GenericOrder` – Type.

The `GenericOrder` can be used to test APIs for their support of generic ordered types.

`Test.GenericSet` – Type.

The `GenericSet` can be used to test generic set APIs that program to the `AbstractSet` interface, in order to ensure that functions can work with set types besides the standard `Set` and `BitSet` types.

`Test.GenericString` – Type.

The `GenericString` can be used to test generic string APIs that program to the `AbstractString` interface, in order to ensure that functions can work with string types besides the standard `String` type.

`Test.detect_ambiguities` – Function.

```
detect_ambiguities(mod1, mod2...; recursive=false,
                                  ambiguous_bottom=false,
                                  allowed_undefineds=nothing)
```

Return a vector of `(Method,Method)` pairs of ambiguous methods defined in the specified modules. Use `recursive=true` to test in all submodules.

`ambiguous_bottom` controls whether ambiguities triggered only by `Union{}` type parameters are included; in most cases you probably want to set this to `false`. See `Base.isambiguous`.

See `Test.detect_unbound_args` for an explanation of `allowed_undefineds`.

Julia 1.8

`allowed_undefineds` requires at least Julia 1.8.

`Test.detect_unbound_args` – Function.

```
detect_unbound_args(mod1, mod2...; recursive=false, allowed_undefineds=nothing)
```

Return a vector of `Methods` which may have unbound type parameters. Use `recursive=true` to test in all submodules.

By default, any undefined symbols trigger a warning. This warning can be suppressed by supplying a collection of `GlobalRefs` for which the warning can be skipped. For example, setting

```
allowed_undefineds = Set([GlobalRef(Base, :active_repl),
                          GlobalRef(Base, :active_repl_backend)])
```

would suppress warnings about `Base.active_repl` and `Base.active_repl_backend`.

Julia 1.8

`allowed_undefineds` requires at least Julia 1.8.

106.10 Workflow for Testing Packages

Using the tools available to us in the previous sections, here is a potential workflow of creating a package and adding tests to it.

Generating an Example Package

For this workflow, we will create a package called `Example`:

```
pkg> generate Example
shell> cd Example
shell> mkdir test
pkg> activate .
```

Creating Sample Functions

The number one requirement for testing a package is to have functionality to test. For that, we will add some simple functions to Example that we can test. Add the following to `src/Example.jl`:

```
module Example

export greet, simple_add, type_multiply

function greet()
    "Hello world!"
end

function simple_add(a, b)
    a + b
end

function type_multiply(a::Float64, b::Float64)
    a * b
end

end
```

Creating a Test Environment

From within the root of the Example package, navigate to the `test` directory, activate a new environment there, and add the Test package to the environment:

```
shell> cd test
pkg> activate .
(test) pkg> add Test
```

Testing Our Package

Now, we are ready to add tests to Example. It is standard practice to create a file within the `test` directory called `runtests.jl` which contains the test sets we want to run. Go ahead and create that file within the `test` directory and add the following code to it:

```
using Example
using Test

@testset "Example tests" begin

    @testset "Math tests" begin
        include("math_tests.jl")
    end

    @testset "Greeting tests" begin
        include("greeting_tests.jl")
    end
end
```

We will need to create those two included files, `math_tests.jl` and `greeting_tests.jl`, and add some tests to them.

> **Note:** Notice how we did not have to specify add `Example` into the `test` environment's `Project.toml`. This is a benefit of Julia's testing system that you could read about more here.

Writing Tests for `math_tests.jl`

Using our knowledge of `Test.jl`, here are some example tests we could add to `math_tests.jl`:

```
@testset "Testset 1" begin
    @test 2 == simple_add(1, 1)
    @test 3.5 == simple_add(1, 2.5)
    @test_throws MethodError simple_add(1, "A")
    @test_throws MethodError simple_add(1, 2, 3)
end

@testset "Testset 2" begin
    @test 1.0 == type_multiply(1.0, 1.0)
    @test isa(type_multiply(2.0, 2.0), Float64)
    @test_throws MethodError type_multiply(1, 2.5)
end
```

Writing Tests for `greeting_tests.jl`

Using our knowledge of `Test.jl`, here are some example tests we could add to `greeting_tests.jl`:

```
@testset "Testset 3" begin
    @test "Hello world!" == greet()
    @test_throws MethodError greet("Antonia")
end
```

Testing Our Package

Now that we have added our tests and our `runtests.jl` script in `test`, we can test our `Example` package by going back to the root of the `Example` package environment and reactivating the `Example` environment:

```
shell> cd ..
pkg> activate .
```

From there, we can finally run our test suite as follows:

```
(Example) pkg> test
     Testing Example
      Status `/tmp/jl_Yngpvy/Project.toml`
  [fa318bd2] Example v0.1.0 `/home/src/Projects/tmp/errata/Example`
  [8dfed614] Test `@stdlib/Test`
      Status `/tmp/jl_Yngpvy/Manifest.toml`
  [fa318bd2] Example v0.1.0 `/home/src/Projects/tmp/errata/Example`
```

```
  [2a0f44e3] Base64 `@stdlib/Base64`
  [b77e0a4c] InteractiveUtils `@stdlib/InteractiveUtils`
  [56ddb016] Logging `@stdlib/Logging`
  [d6f4376e] Markdown `@stdlib/Markdown`
  [9a3f8284] Random `@stdlib/Random`
  [ea8e919c] SHA `@stdlib/SHA`
  [9e88b42a] Serialization `@stdlib/Serialization`
  [8dfed614] Test `@stdlib/Test`
     Testing Running tests...
Test Summary: | Pass  Total
Example tests |    9      9
     Testing Example tests passed
```

And if all went correctly, you should see a similar output as above. Using `Test.jl`, more complicated tests can be added for packages but this should ideally point developers in the direction of how to get started with testing their own created packages.

Code Coverage

Code coverage tracking during tests can be enabled using the `pkg> test --coverage` flag (or at a lower level using the `--code-coverage` julia arg). This is on by default in the julia-runtest GitHub action.

To evaluate coverage either manually inspect the `.cov` files that are generated beside the source files locally, or in CI use the julia-processcoverage GitHub action.

Julia 1.11

Since Julia 1.11, coverage is not collected during the package precompilation phase.

Chapter 107

UUIDs

`UUIDs.uuid1` – Function.

```
uuid1([rng::AbstractRNG])::UUID
```

Generates a version 1 (time-based) universally unique identifier (UUID), as specified by RFC 4122. Note that the Node ID is randomly generated (does not identify the host) according to section 4.5 of the RFC.

The default rng used by `uuid1` is not `Random.default_rng()` and every invocation of `uuid1()` without an argument should be expected to return a unique identifier. Importantly, the outputs of `uuid1` do not repeat even when `Random.seed!(seed)` is called. Currently (as of Julia 1.6), `uuid1` uses `Random.RandomDevice` as the default rng. However, this is an implementation detail that may change in the future.

Julia 1.6

The output of `uuid1` does not depend on `Random.default_rng()` as of Julia 1.6.

Examples

```
julia> using Random

julia> rng = MersenneTwister(1234);

julia> uuid1(rng)
UUID("cfc395e8-590f-11e8-1f13-43a2532b2fa8")
```

`UUIDs.uuid4` – Function.

```
uuid4([rng::AbstractRNG])::UUID
```

Generates a version 4 (random or pseudo-random) universally unique identifier (UUID), as specified by RFC 4122.

The default rng used by `uuid4` is not `Random.default_rng()` and every invocation of `uuid4()` without an argument should be expected to return a unique identifier. Importantly, the outputs of `uuid4` do not repeat even when `Random.seed!(seed)` is called. Currently (as of Julia 1.6), `uuid4` uses `Random.RandomDevice` as the default rng. However, this is an implementation detail that may change in the future.

Julia 1.6

The output of uuid4 does not depend on Random.default_rng() as of Julia 1.6.

Examples

```
julia> using Random

julia> rng = Xoshiro(123);

julia> uuid4(rng)
UUID("856e446e-0c6a-472a-9638-f7b8557cd282")
```

UUIDs.uuid5 – Function.

```
uuid5(ns::UUID, name::String)::UUID
```

Generates a version 5 (namespace and domain-based) universally unique identifier (UUID), as specified by RFC 4122.

Julia 1.1

This function requires at least Julia 1.1.

Examples

```
julia> using Random

julia> rng = Xoshiro(123);

julia> u4 = uuid4(rng)
UUID("856e446e-0c6a-472a-9638-f7b8557cd282")

julia> u5 = uuid5(u4, "julia")
UUID("2df91e3f-da06-5362-a6fe-03772f2e14c9")
```

UUIDs.uuid_version – Function.

```
uuid_version(u::UUID)::Int
```

Inspects the given UUID and returns its version (see RFC 4122).

Examples

```
julia> uuid_version(uuid4())
4
```

Part IV

Developer Documentation

Chapter 108

Documentation of Julia's Internals

108.1 Initialization of the Julia runtime

How does the Julia runtime execute `julia -e 'println("Hello World!")'` ?

`main()`

Execution starts at `main()` in `cli/loader_exe.c`, which calls `jl_load_repl()` in `cli/loader_lib.c` which loads a few libraries, eventually calling `jl_repl_entrypoint()` in `src/jlapi.c`.

`jl_repl_entrypoint()` calls `libsupport_init()` to set the C library locale and to initialize the "ios" library (see `ios_init_stdstreams()` and Legacy `ios.c` library).

Next `jl_parse_opts()` is called to process command line options. Note that `jl_parse_opts()` only deals with options that affect code generation or early initialization. Other options are handled later by `exec_options()` in `base/client.jl`.

`jl_parse_opts()` stores command line options in the global `jl_options` struct.

`julia_init()`

`julia_init()` in `init.c` is called by `main()` and calls `_julia_init()` in `init.c`.

`_julia_init()` begins by calling `libsupport_init()` again (it does nothing the second time).

`restore_signals()` is called to zero the signal handler mask.

`jl_resolve_sysimg_location()` searches configured paths for the base system image. See Building the Julia system image.

`jl_gc_init()` sets up allocation pools and lists for weak refs, preserved values and finalization.

`jl_init_frontend()` loads and initializes a pre-compiled femtolisp image containing the scanner/parser.

`jl_init_types()` creates `jl_datatype_t` type description objects for the built-in types defined in `julia.h`. e.g.

```
jl_any_type = jl_new_abstracttype(jl_symbol("Any"), core, NULL, jl_emptysvec);
jl_any_type->super = jl_any_type;
```

```
jl_type_type = jl_new_abstracttype(jl_symbol("Type"), core, jl_any_type, jl_emptysvec);

jl_int32_type = jl_new_primitivetype(jl_symbol("Int32"), core,
                                     jl_any_type, jl_emptysvec, 32);
```

`jl_init_tasks()` creates the `jl_datatype_t* jl_task_type` object; initializes the global `jl_root_task` struct; and sets `jl_current_task` to the root task.

`jl_init_codegen()` initializes the LLVM library.

`jl_init_serializer()` initializes 8-bit serialization tags for builtin `jl_value_t` values.

If there is no sysimg file (`!jl_options.image_file`) then the `Core` and `Main` modules are created and `boot.jl` is evaluated:

`jl_core_module = jl_new_module(jl_symbol("Core"), NULL)` creates the Julia `Core` module.

`jl_init_intrinsic_functions()` creates a new Julia module `Intrinsics` containing constant `jl_intrinsic_type` symbols. These define an integer code for each intrinsic function. `emit_intrinsic()` translates these symbols into LLVM instructions during code generation.

`jl_init_primitives()` hooks C functions up to Julia function symbols. e.g. the symbol `Core.:(===)()` is bound to C function pointer `jl_f_is()` by calling `add_builtin_func("===", jl_f_is)`.

`jl_new_main_module()` creates the global "Main" module and sets `jl_current_task->current_module = jl_main_module`.

Note: `_julia_init()` then sets `jl_root_task->current_module = jl_core_module`. `jl_root_task` is an alias of `jl_current_task` at this point, so the `current_module` set by `jl_new_main_module()` above is overwritten.

`jl_load("boot.jl", sizeof("boot.jl"))` calls `jl_parse_eval_all` which repeatedly calls `jl_toplevel_eval_flex()` to execute `boot.jl`. <!– TODO – drill down into eval? –>

`jl_get_builtin_hooks()` initializes global C pointers to Julia globals defined in `boot.jl`.

`jl_init_box_caches()` pre-allocates global boxed integer value objects for values up to 1024. This speeds up allocation of boxed ints later on. e.g.:

```
jl_value_t *jl_box_uint8(uint32_t x)
{
    return boxed_uint8_cache[(uint8_t)x];
}
```

`_julia_init()` iterates over the `jl_core_module->bindings.table` looking for `jl_datatype_t` values and sets the type name's module prefix to `jl_core_module`.

`jl_add_standard_imports(jl_main_module)` does "using Base" in the "Main" module.

Note: `_julia_init()` now reverts to `jl_root_task->current_module = jl_main_module` as it was before being set to `jl_core_module` above.

Platform specific signal handlers are initialized for `SIGSEGV` (OSX, Linux), and `SIGFPE` (Windows).

Other signals (`SIGINFO`, `SIGBUS`, `SIGILL`, `SIGTERM`, `SIGABRT`, `SIGQUIT`, `SIGSYS` and `SIGPIPE`) are hooked up to `sigdie_handler()` which prints a backtrace.

jl_init_restored_module() calls jl_module_run_initializer() for each deserialized module to run the __init__() function.

Finally sigint_handler() is hooked up to SIGINT and calls jl_throw(jl_interrupt_exception).

_julia_init() then returns back to main() in cli/loader_exe.c and main() calls repl_entrypoint(argc, (char**)argv).

sysimg

If there is a sysimg file, it contains a pre-cooked image of the Core and Main modules (and whatever else is created by boot.jl). See Building the Julia system image.

jl_restore_system_image() deserializes the saved sysimg into the current Julia runtime environment and initialization continues after jl_init_box_caches() below...

Note: jl_restore_system_image() (and staticdata.c in general) uses the Legacy ios.c library.

repl_entrypoint()

repl_entrypoint() loads the contents of argv[] into Base.ARGS.

If a .jl "program" file was supplied on the command line, then exec_program() calls jl_load(program,len) which calls jl_parse_eval_all which repeatedly calls jl_toplevel_eval_flex() to execute the program.

However, in our example (julia -e 'println("Hello World!")'), jl_get_global(jl_base_module, jl_symbol("_start")) looks up Base._start and jl_apply() executes it.

Base._start

Base._start calls Base.exec_options which calls jl_parse_input_line("println("Hello World!")") to create an expression object and Core.eval(Main, ex) to execute the parsed expression ex in the module context of Main.

Core.eval

Core.eval(Main, ex) calls jl_toplevel_eval_in(m, ex), which calls jl_toplevel_eval_flex. jl_toplevel_eval_flex implements a simple heuristic to decide whether to compile a given code thunk or run it by interpreter. When given println("Hello World!"), it would usually decide to run the code by interpreter, in which case it calls jl_interpret_toplevel_thunk, which then calls eval_body.

The stack dump below shows how the interpreter works its way through various methods of Base.println() and Base.print() before arriving at write(s::IO, a::Array{T}) where T which does ccall(jl_uv_write()).

jl_uv_write() calls uv_write() to write "Hello World!" to JL_STDOUT. See Libuv wrappers for stdio.:

```
Hello World!
```

Since our example has just one function call, which has done its job of printing "Hello World!", the stack now rapidly unwinds back to main().

Stack frame	Source code	Notes
jl_uv_write()	jl_uv.c	called though ccall
julia_write_282942	stream.jl	function write!(s::IO, a::Array{T}) where T
julia_print_284639	ascii.jl	print(io::IO, s::String) = (write(io, s); nothing)
jlcall_print_284639		
jl_apply()	julia.h	
jl_trampoline()	builtins.c	
jl_apply()	julia.h	
jl_apply_generic()	gf.c	Base.print(Base.TTY, String)
jl_apply()	julia.h	
jl_trampoline()	builtins.c	
jl_apply()	julia.h	
jl_apply_generic()	gf.c	Base.print(Base.TTY, String, Char, Char...)
jl_apply()	julia.h	
jl_f_apply()	builtins.c	
jl_apply()	julia.h	
jl_trampoline()	builtins.c	
jl_apply()	julia.h	
jl_apply_generic()	gf.c	Base.println(Base.TTY, String, String...)
jl_apply()	julia.h	
jl_trampoline()	builtins.c	
jl_apply()	julia.h	
jl_apply_generic()	gf.c	Base.println(String,)
jl_apply()	julia.h	
do_call()	interpreter.c	
eval_body()	interpreter.c	
jl_interpret_toplevel_thunk	interpreter.c	
jl_toplevel_eval_flex	toplevel.c	
jl_toplevel_eval_in	toplevel.c	
Core.eval	boot.jl	

jl_atexit_hook()

main() calls jl_atexit_hook(). This calls Base._atexit, then calls jl_gc_run_all_finalizers() and cleans up libuv handles.

julia_save()

Finally, main() calls julia_save(), which if requested on the command line, saves the runtime state to a new system image. See jl_compile_all() and jl_save_system_image().

108.2 Julia ASTs

Julia has two representations of code. First there is a surface syntax AST returned by the parser (e.g. the Meta.parse function), and manipulated by macros. It is a structured representation of code as it is written, constructed by julia-parser.scm from a character stream. Next there is a lowered form, or IR (intermediate representation), which is used by type inference and code generation. In the lowered

form there are fewer types of nodes, all macros are expanded, and all control flow is converted to explicit branches and sequences of statements. The lowered form is constructed by `julia-syntax.scm`.

First we will focus on the AST, since it is needed to write macros.

Surface syntax AST

Front end ASTs consist almost entirely of `Expr`s and atoms (e.g. symbols, numbers). There is generally a different expression head for each visually distinct syntactic form. Examples will be given in s-expression syntax. Each parenthesized list corresponds to an Expr, where the first element is the head. For example `(call f x)` corresponds to `Expr(:call, :f, :x)` in Julia.

Calls

Input	AST
`f(x)`	`(call f x)`
`f(x, y=1, z=2)`	`(call f x (kw y 1) (kw z 2))`
`f(x; y=1)`	`(call f (parameters (kw y 1)) x)`
`f(x...)`	`(call f (... x))`

do syntax:

```
f(x) do a,b
    body
end
```

parses as `(do (call f x) (-> (tuple a b) (block body)))`.

Operators

Most uses of operators are just function calls, so they are parsed with the head `call`. However some operators are special forms (not necessarily function calls), and in those cases the operator itself is the expression head. In julia-parser.scm these are referred to as "syntactic operators". Some operators (+ and *) use N-ary parsing; chained calls are parsed as a single N-argument call. Finally, chains of comparisons have their own special expression structure.

Bracketed forms

Macros

Strings

Doc string syntax:

```
"some docs"
f(x) = x
```

parses as `(macrocall (|.| Core '@doc) (line) "some docs" (= (call f x) (block x)))`.

Input	AST
`x+y`	`(call + x y)`
`a+b+c+d`	`(call + a b c d)`
`2x`	`(call * 2 x)`
`a&&b`	`(&& a b)`
`x += 1`	`(+= x 1)`
`a ? 1 : 2`	`(if a 1 2)`
`a,b`	`(tuple a b)`
`a==b`	`(call == a b)`
`1<i<=n`	`(comparison 1 < i <= n)`
`a.b`	`(. a (quote b))`
`a.(b)`	`(. a (tuple b))`

Input	AST
`a[i]`	`(ref a i)`
`t[i;j]`	`(typed_vcat t i j)`
`t[i j]`	`(typed_hcat t i j)`
`t[a b; c d]`	`(typed_vcat t (row a b) (row c d))`
`t[a b;;; c d]`	`(typed_ncat t 3 (row a b) (row c d))`
`a{b}`	`(curly a b)`
`a{b;c}`	`(curly a (parameters c) b)`
`[x]`	`(vect x)`
`[x,y]`	`(vect x y)`
`[x;y]`	`(vcat x y)`
`[x y]`	`(hcat x y)`
`[x y; z t]`	`(vcat (row x y) (row z t))`
`[x;y;; z;t;;;]`	`(ncat 3 (nrow 2 (nrow 1 x y) (nrow 1 z t)))`
`[x for y in z, a in b]`	`(comprehension (generator x (= y z) (= a b)))`
`T[x for y in z]`	`(typed_comprehension T (generator x (= y z)))`
`(a, b, c)`	`(tuple a b c)`
`(a; b; c)`	`(block a b c)`

Input	AST
`@m x y`	`(macrocall @m (line) x y)`
`Base.@m x y`	`(macrocall (. Base (quote @m)) (line) x y)`
`@Base.m x y`	`(macrocall (. Base (quote @m)) (line) x y)`

Imports and such

`using` has the same representation as `import`, but with expression head `:using` instead of `:import`.

To programmatically create a `public` statement, you can use `Expr(:public, :a, :b)` or, closer to regular code, `Meta.parse("public a, b")`. This approach is necessary due to current limitations on `public`. The `public` keyword is only recognized at the syntactic top level within a file (`parse_stmts`) or module. This restriction was implemented to prevent breaking existing code that used `public` as an identifier when it was introduced in Julia 1.11.

Input	AST
`"a"`	`"a"`
`x"y"`	`(macrocall @x_str (line) "y")`
`x"y"z`	`(macrocall @x_str (line) "y" "z")`
`"x = $x"`	`(string "x = " x)`
`` `a b c` ``	`(macrocall @cmd (line) "a b c")`

Input	AST
`import a`	`(import (. a))`
`import a.b.c`	`(import (. a b c))`
`import ...a`	`(import (. . . . a))`
`import a.b, c.d`	`(import (. a b) (. c d))`
`import Base: x`	`(import (: (. Base) (. x)))`
`import Base: x, y`	`(import (: (. Base) (. x) (. y)))`
`export a, b`	`(export a b)`
`public a, b`	`(public a b)`

Numbers

Julia supports more number types than many scheme implementations, so not all numbers are represented directly as scheme numbers in the AST.

Input	AST
`11111111111111111111`	`(macrocall @int128_str nothing "11111111111111111111")`
`0xffffffffffffffff`	`(macrocall @uint128_str nothing "0xffffffffffffffff")`
`1111...many digits...`	`(macrocall @big_str nothing "1111....")`

Block forms

A block of statements is parsed as `(block stmt1 stmt2 ...)`.

If statement:

```
if a
    b
elseif c
    d
else
    e
end
```

parses as:

```
(if a (block (line 2) b)
    (elseif (block (line 3) c) (block (line 4) d)
            (block (line 6) e)))
```

A `while` loop parses as `(while condition body)`.

A `for` loop parses as `(for (= var iter) body)`. If there is more than one iteration specification, they are parsed as a block: `(for (block (= v1 iter1) (= v2 iter2)) body)`.

`break` and `continue` are parsed as 0-argument expressions `(break)` and `(continue)`.

`let` is parsed as `(let (= var val) body)` or `(let (block (= var1 val1) (= var2 val2) ...) body)`, like `for` loops.

A basic function definition is parsed as `(function (call f x) body)`. A more complex example:

```
function f(x::T; k = 1) where T
    return x+1
end
```

parses as:

```
(function (where (call f (parameters (kw k 1))
                         (:: x T))
                 T)
          (block (line 2) (return (call + x 1))))
```

Type definition:

```
mutable struct Foo{T<:S}
    x::T
end
```

parses as:

```
(struct true (curly Foo (<: T S))
        (block (line 2) (:: x T)))
```

The first argument is a boolean telling whether the type is mutable.

`try` blocks parse as `(try try_block var catch_block finally_block)`. If no variable is present after `catch`, `var` is `#f`. If there is no `finally` clause, then the last argument is not present.

Quote expressions

Julia source syntax forms for code quoting (`quote` and `:( )`) support interpolation with `$`. In Lisp terminology, this means they are actually "backquote" or "quasiquote" forms. Internally, there is also a need for code quoting without interpolation. In Julia's scheme code, non-interpolating quote is represented with the expression head `inert`.

`inert` expressions are converted to Julia `QuoteNode` objects. These objects wrap a single value of any type, and when evaluated simply return that value.

A `quote` expression whose argument is an atom also gets converted to a `QuoteNode`.

Line numbers

Source location information is represented as `(line line_num file_name)` where the third component is optional (and omitted when the current line number, but not file name, changes).

These expressions are represented as `LineNumberNode`s in Julia.

Macros

Macro hygiene is represented through the expression head pair `escape` and `hygienic-scope`. The result of a macro expansion is automatically wrapped in `(hygienic-scope block module [lno])`, to represent the result of the new scope. The user can insert `(escape block)` inside to interpolate code from the caller. The lno is the `__source__` argument of the macro, if included.

Lowered form

Lowered form (IR) is more important to the compiler, since it is used for type inference, optimizations like inlining, and code generation. It is also less obvious to the human, since it results from a significant rearrangement of the input syntax.

In addition to `Symbol`s and some number types, the following data types exist in lowered form:

- `Expr`

  Has a node type indicated by the head field, and an `args` field which is a `Vector{Any}` of subexpressions. While almost every part of a surface AST is represented by an `Expr`, the IR uses only a limited number of `Expr`s, mostly for calls and some top-level-only forms.

- `SlotNumber`

  Identifies arguments and local variables by consecutive numbering. It has an integer-valued `id` field giving the slot index. The types of these slots can be found in the `slottypes` field of their `CodeInfo` object.

- `Argument`

  The same as `SlotNumber`, but appears only post-optimization. Indicates that the referenced slot is an argument of the enclosing function.

- `CodeInfo`

  Wraps the IR of a group of statements. Its `code` field is an array of expressions to execute.

- `GotoNode`

  Unconditional branch. The argument is the branch target, represented as an index in the code array to jump to.

- `GotoIfNot`

  Conditional branch. If the `cond` field evaluates to false, goes to the index identified by the `dest` field.

- `ReturnNode`

  Returns its argument (the `val` field) as the value of the enclosing function. If the `val` field is undefined, then this represents an unreachable statement.

- `QuoteNode`

  Wraps an arbitrary value to reference as data. For example, the function `f() = :a` contains a `QuoteNode` whose `value` field is the symbol `a`, in order to return the symbol itself instead of evaluating it.

- `GlobalRef`

  Refers to global variable `name` in module `mod`.

- `SSAValue`

  Refers to a consecutively-numbered (starting at 1) static single assignment (SSA) variable inserted by the compiler. The number (`id`) of an `SSAValue` is the code array index of the expression whose value it represents.

- `NewvarNode`

  Marks a point where a variable (slot) is created. This has the effect of resetting a variable to undefined.

`Expr` types

These symbols appear in the head field of `Expr`s in lowered form.

- `call`

  Function call (dynamic dispatch). `args[1]` is the function to call, `args[2:end]` are the arguments.

- `invoke`

  Function call (static dispatch). `args[1]` is the MethodInstance to call, `args[2:end]` are the arguments (including the function that is being called, at `args[2]`).

- `static_parameter`

  Reference a static parameter by index.

- `=`

  Assignment. In the IR, the first argument is always a `SlotNumber` or a `GlobalRef`.

- `method`

  Adds a method to a generic function and assigns the result if necessary.

  Has a 1-argument form and a 3-argument form. The 1-argument form arises from the syntax `function foo end`. In the 1-argument form, the argument is a symbol. If this symbol already names a function in the current scope, nothing happens. If the symbol is undefined, a new function is created and assigned to the identifier specified by the symbol. If the symbol is defined but names a non-function, an error is raised. The definition of "names a function" is that the binding is constant, and refers to an object of singleton type. The rationale for this is that an instance of a singleton type uniquely identifies the type to add the method to. When the type has fields, it wouldn't be clear whether the method was being added to the instance or its type.

  The 3-argument form has the following arguments:

- `args[1]`

  A function name, or `nothing` if unknown or unneeded. If a symbol, then the expression first behaves like the 1-argument form above. This argument is ignored from then on. It can be `nothing` when methods are added strictly by type, `(::T)(x) = x`, or when a method is being added to an existing function, `MyModule.f(x) = x`.

- `args[2]`

  A `SimpleVector` of argument type data. `args[2][1]` is a `SimpleVector` of the argument types, and `args[2][2]` is a `SimpleVector` of type variables corresponding to the method's static parameters.

- `args[3]`

  A `CodeInfo` of the method itself. For "out of scope" method definitions (adding a method to a function that also has methods defined in different scopes) this is an expression that evaluates to a `:lambda` expression.

- `struct_type`

  A 7-argument expression that defines a new `struct`:

  - `args[1]`

    The name of the `struct`

  - `args[2]`

    A `call` expression that creates a `SimpleVector` specifying its parameters

  - `args[3]`

    A `call` expression that creates a `SimpleVector` specifying its fieldnames

  - `args[4]`

    A `Symbol`, `GlobalRef`, or `Expr` specifying the supertype (e.g., `:Integer`, `GlobalRef(Core, :Any)`, or `:(Core.apply_type(AbstractArray, T, N))`)

  - `args[5]`

    A `call` expression that creates a `SimpleVector` specifying its fieldtypes

  - `args[6]`

    A Bool, true if `mutable`

  - `args[7]`

    The number of arguments to initialize. This will be the number of fields, or the minimum number of fields called by an inner constructor's new statement.

- `abstract_type`

  A 3-argument expression that defines a new abstract type. The arguments are the same as arguments 1, 2, and 4 of `struct_type` expressions.

- `primitive_type`

  A 4-argument expression that defines a new primitive type. Arguments 1, 2, and 4 are the same as `struct_type`. Argument 3 is the number of bits.

  Julia 1.5

  `struct_type`, `abstract_type`, and `primitive_type` were removed in Julia 1.5 and replaced by calls to new builtins.

- `global`

  Declares a global binding.

- `const`

  Declares a (global) variable as constant.

- `new`

  Allocates a new struct-like object. First argument is the type. The `new` pseudo-function is lowered to this, and the type is always inserted by the compiler. This is very much an internal-only feature, and does no checking. Evaluating arbitrary `new` expressions can easily segfault.

- `splatnew`

  Similar to `new`, except field values are passed as a single tuple. Works similarly to `splat(new)` if `new` were a first-class function, hence the name.

- `isdefined`

  `Expr(:isdefined, :x)` returns a Bool indicating whether `x` has already been defined in the current scope.

- `the_exception`

  Yields the caught exception inside a `catch` block, as returned by `jl_current_exception(ct)`.

- `enter`

  Enters an exception handler (`setjmp`). `args[1]` is the label of the catch block to jump to on error. Yields a token which is consumed by `pop_exception`.

- `leave`

  Pop exception handlers. `args[1]` is the number of handlers to pop.

- `pop_exception`

  Pop the stack of current exceptions back to the state at the associated `enter` when leaving a catch block. `args[1]` contains the token from the associated `enter`.

  Julia 1.1

  `pop_exception` is new in Julia 1.1.

- `inbounds`

  Controls turning bounds checks on or off. A stack is maintained; if the first argument of this expression is true or false (`true` means bounds checks are disabled), it is pushed onto the stack. If the first argument is `:pop`, the stack is popped.

- `boundscheck`

  Has the value `false` if inlined into a section of code marked with `@inbounds`, otherwise has the value `true`.

- `loopinfo`

  Marks the end of the a loop. Contains metadata that is passed to `LowerSimdLoop` to either mark the inner loop of `@simd` expression, or to propagate information to LLVM loop passes.

- `copyast`

  Part of the implementation of quasi-quote. The argument is a surface syntax AST that is simply copied recursively and returned at run time.

- `meta`

  Metadata. `args[1]` is typically a symbol specifying the kind of metadata, and the rest of the arguments are free-form. The following kinds of metadata are commonly used:

  - `:inline` and `:noinline`: Inlining hints.

- `foreigncall`

  Statically-computed container for `ccall` information. The fields are:

  - `args[1]` : name

    The expression that'll be parsed for the foreign function.

  - `args[2]::Type` : RT

    The (literal) return type, computed statically when the containing method was defined.

  - `args[3]::SimpleVector` (of Types) : AT

    The (literal) vector of argument types, computed statically when the containing method was defined.

  - `args[4]::Int` : nreq

    The number of required arguments for a varargs function definition.

  - `args[5]::QuoteNode{<:Union{Symbol,Tuple{Symbol,UInt16}, Tuple{Symbol,UInt16,Bool}}`: calling convention

    The calling convention for the call, optionally with effects, and `gc_safe` (safe to execute concurrently to GC.).

  - `args[6:5+length(args[3])]` : arguments

    The values for all the arguments (with types of each given in args[3]).

  - `args[6+length(args[3])+1:end]` : gc-roots

    The additional objects that may need to be gc-rooted for the duration of the call. See Working with LLVM for where these are derived from and how they get handled.

- `new_opaque_closure`

  Constructs a new opaque closure. The fields are:

  - `args[1]` : signature

    The function signature of the opaque closure. Opaque closures don't participate in dispatch, but the input types can be restricted.

  - `args[2]` : lb

    Lower bound on the output type. (Defaults to `Union{}`)

  - `args[3]` : ub

    Upper bound on the output type. (Defaults to `Any`)

- `args[4]` : constprop

  Indicates whether the opaque closure's identity may be used for constant propagation. The `@opaque` macro enables this by default, but this will cause additional inference which may be undesirable and prevents the code from running during precompile. If `args[4]` is a method, the argument is considered skipped.

- `args[5]` : method

  The actual method as an `opaque_closure_method` expression.

- `args[6:end]` : captures

  The values captured by the opaque closure.

Julia 1.7

Opaque closures were added in Julia 1.7

Method

A unique'd container describing the shared metadata for a single method.

- `name`, `module`, `file`, `line`, `sig`

  Metadata to uniquely identify the method for the computer and the human.

- `ambig`

  Cache of other methods that may be ambiguous with this one.

- `specializations`

  Cache of all MethodInstance ever created for this Method, used to ensure uniqueness. Uniqueness is required for efficiency, especially for incremental precompile and tracking of method invalidation.

- `source`

  The original source code (if available, usually compressed).

- `generator`

  A callable object which can be executed to get specialized source for a specific method signature.

- `roots`

  Pointers to non-AST things that have been interpolated into the AST, required by compression of the AST, type-inference, or the generation of native code.

- `nargs`, `isva`, `called`, `is_for_opaque_closure`,

  Descriptive bit-fields for the source code of this Method.

- `primary_world`

  The world age that "owns" this Method.

MethodInstance

A unique'd container describing a single callable signature for a Method. See especially Proper maintenance and care of multi-threading locks for important details on how to modify these fields safely.

- `specTypes`

  The primary key for this MethodInstance. Uniqueness is guaranteed through a `def.specializations` lookup.

- `def`

  The `Method` that this function describes a specialization of. Or a `Module`, if this is a top-level Lambda expanded in Module, and which is not part of a Method.

- `sparam_vals`

  The values of the static parameters in `specTypes`. For the `MethodInstance` at `Method.unspecialized`, this is the empty `SimpleVector`. But for a runtime `MethodInstance` from the `MethodTable` cache, this will always be defined and indexable.

- `backedges`

  We store the reverse-list of cache dependencies for efficient tracking of incremental reanalysis/recompilation work that may be needed after a new method definitions. This works by keeping a list of the other `MethodInstance` that have been inferred or optimized to contain a possible call to this `MethodInstance`. Those optimization results might be stored somewhere in the cache, or it might have been the result of something we didn't want to cache, such as constant propagation. Thus we merge all of those backedges to various cache entries here (there's almost always only the one applicable cache entry with a sentinel value for max_world anyways).

- `cache`

  Cache of `CodeInstance` objects that share this template instantiation.

CodeInstance

- `def`

  The `MethodInstance` that this cache entry is derived from.

- `owner`

  A token that represents the owner of this `CodeInstance`. Will use `jl_egal` to match.

- `rettype/rettype_const`

  The inferred return type for the `specFunctionObject` field, which (in most cases) is also the computed return type for the function in general.

- `inferred`

  May contain a cache of the inferred source for this function, or it could be set to `nothing` to just indicate `rettype` is inferred.

- `ftpr`

  The generic jlcall entry point.

- `jlcall_api`

  The ABI to use when calling `fptr`. Some significant ones include:

  - 0 - Not compiled yet
  - 1 - JL_CALLABLE `jl_value_t *(*)(jl_value_t *f, jl_value_t *args[nargs], uint32_t nargs)`
  - 2 - Constant (value stored in `rettype_const`)
  - 3 - With Static-parameters forwarded `jl_value_t *(*)(jl_svec_t *sparams, jl_value_t *f, jl_value_t *args[nargs], uint32_t nargs)`
  - 4 - Run in interpreter `jl_value_t *(*)(jl_method_instance_t *meth, jl_value_t *f, jl_value_t *args[nargs], uint32_t nargs)`

- `min_world` / `max_world`

  The range of world ages for which this method instance is valid to be called. If max_world is the special token value `-1`, the value is not yet known. It may continue to be used until we encounter a backedge that requires us to reconsider.

- Timing fields

  - `time_infer_total`: Total cost of computing `inferred` originally as wall-time from start to finish.
  - `time_infer_cache_saved`: The cost saved from `time_infer_total` by having caching. Adding this to `time_infer_total` should give a stable estimate for comparing the cost of two implementations or one implementation over time. This is generally an over-estimate of the time to infer something, since the cache is frequently effective at handling repeated work.
  - `time_infer_self`: Self cost of julia inference for `inferred` (a portion of `time_infer_total`). This is simply the incremental cost of compiling this one method, if given a fully populated cache of all call targets, even including constant inference results and LimitedAccuracy results, which generally are not in a cache.
  - `time_compile`: Self cost of llvm JIT compilation (e.g. of computing `invoke` from `inferred`). A total cost estimate can be computed by walking all of the edges contents and summing those, while accounting for cycles and duplicates. (This field currently does not include any measured AOT compile times.)

CodeInfo

A (usually temporary) container for holding lowered (and possibly inferred) source code.

- `code`

  An `Any` array of statements

- `slotnames`

  An array of symbols giving names for each slot (argument or local variable).

- `slotflags`

  A `UInt8` array of slot properties, represented as bit flags:

  - 0x02 - assigned (only false if there are *no* assignment statements with this var on the left)
  - 0x08 - used (if there is any read or write of the slot)
  - 0x10 - statically assigned once
  - 0x20 - might be used before assigned. This flag is only valid after type inference.

- `ssavaluetypes`

  Either an array or an `Int`.

  If an `Int`, it gives the number of compiler-inserted temporary locations in the function (the length of `code` array). If an array, specifies a type for each location.

- `ssaflags`

  Statement-level 32 bits flags for each expression in the function. See the definition of `jl_code_info_t` in julia.h for more details.

These are only populated after inference (or by generated functions in some cases):

- `debuginfo`

  An object to retrieve source information for each statements, see How to interpret line numbers in a `CodeInfo` object.

- `rettype`

  The inferred return type of the lowered form (IR). Default value is `Any`. This is mostly present for convenience, as (due to the way OpaqueClosures work) it is not necessarily the rettype used by codegen.

- `parent`

  The `MethodInstance` that "owns" this object (if applicable).

- `edges`

  Forward edges to method instances that must be invalidated.

- `min_world/max_world`

  The range of world ages for which this code was valid at the time when it had been inferred.

Optional Fields:

- `slottypes`

  An array of types for the slots.

- `method_for_inference_limit_heuristics`

  The `method_for_inference_heuristics` will expand the given method's generator if necessary during inference.

Boolean properties:

- `propagate_inbounds`

  Whether this should propagate `@inbounds` when inlined for the purpose of eliding `@boundscheck` blocks.

`UInt8` settings:

- `constprop`, `inlineable`
  - 0 = use heuristic
  - 1 = aggressive
  - 2 = none
- `purity` Constructed from 5 bit flags:
  - 0x01 « 0 = this method is guaranteed to return or terminate consistently (`:consistent`)
  - 0x01 « 1 = this method is free from externally semantically visible side effects (`:effect_free`)
  - 0x01 « 2 = this method is guaranteed to not throw an exception (`:nothrow`)
  - 0x01 « 3 = this method is guaranteed to terminate (`:terminates_globally`)
  - 0x01 « 4 = the syntactic control flow within this method is guaranteed to terminate (`:terminates_locally`)

  See the documentation of `Base.@assume_effects` for more details.

How to interpret line numbers in a `CodeInfo` object

There are 2 common forms for this data: one used internally that compresses the data somewhat and one used in the compiler. They contain the same basic info, but the compiler version is all mutable while the version used internally is not.

Many consumers may be able to call `Base.IRShow.buildLineInfoNode`, `Base.IRShow.append_scopes!`, or `Stacktraces.lookup(::InterpreterIP)` to avoid needing to (re-)implement these details specifically.

The definitions of each of these are:

```
struct Core.DebugInfo
    @noinline
    def::Union{Method,MethodInstance,Symbol}
    linetable::Union{Nothing,DebugInfo}
    edges::SimpleVector{DebugInfo}
    codelocs::String # compressed data
end
mutable struct Core.Compiler.DebugInfoStream
    def::Union{Method,MethodInstance,Symbol}
    linetable::Union{Nothing,DebugInfo}
    edges::Vector{DebugInfo}
    firstline::Int32 # the starting line for this block (specified by an index of 0)
    codelocs::Vector{Int32} # for each statement:
        # index into linetable (if defined), else a line number (in the file represented by def)
        # then index into edges
        # then index into edges[linetable]
end
```

- `def` : where this `DebugInfo` was defined (the `Method`, `MethodInstance`, or `Symbol` of file scope, for example)
- `linetable`

  Another `DebugInfo` that this was derived from, which contains the actual line numbers, such that this DebugInfo contains only the indexes into it. This avoids making copies, as well as makes it possible to track how each individual statement transformed from source to optimized, not just the separate line numbers. If `def` is not a Symbol, then that object replaces the current function object for the metadata on what function is conceptually being executed (e.g. think Cassette transforms here). The `codelocs` values described below also are interpreted as an index into the `codelocs` in this object, instead of being a line number itself.
- `edges` : Vector of the unique DebugInfo for every function inlined into this (which recursively have the edges for everything inlined into them).
- `firstline` (when uncompressed to DebugInfoStream)

  The line number associated with the `begin` statement (or other keyword such as `function` or `quote`) that delineates where this code definition "starts".
- `codelocs` (when uncompressed to `DebugInfoStream`)

  A vector of indices, with 3 values for each statement in the IR plus one for the starting point of the block, that describe the stacktrace from that point:

  1. the integer index into the `linetable.codelocs` field, giving the original location associated with each statement (including its syntactic edges), or zero indicating no change to the line number from the previously executed statement (which is not necessarily syntactic or lexical prior), or the line number itself if the `linetable` field is `nothing`.
  2. the integer index into `edges`, giving the `DebugInfo` inlined there, or zero if there are no edges.
  3. (if entry 2 is non-zero) the integer index into `edges[].codelocs`, to interpret recursively for each function in the inlining stack, or zero indicating to use `edges[].firstline` as the line number.

  Special codes include:

  - `(zero, zero, *)`: no change to the line number or edges from the previous statement (you may choose to interpret this either syntactically or lexically). The inlining depth also might have changed, though most callers should ignore that.
  - `(zero, non-zero, *)` : no line number, just edges (usually because of macro-expansion into top-level code).

108.3 More about types

If you've used Julia for a while, you understand the fundamental role that types play. Here we try to get under the hood, focusing particularly on Parametric Types.

Types and sets (and `Any` and `Union{}`/`Bottom`)

It's perhaps easiest to conceive of Julia's type system in terms of sets. While programs manipulate individual values, a type refers to a set of values. This is not the same thing as a collection; for example a `Set` of values is itself a single `Set` value. Rather, a type describes a set of *possible* values, expressing uncertainty about which value we have.

A *concrete* type T describes the set of values whose direct tag, as returned by the `typeof` function, is T. An *abstract* type describes some possibly-larger set of values.

`Any` describes the entire universe of possible values. `Integer` is a subset of `Any` that includes `Int`, `Int8`, and other concrete types. Internally, Julia also makes heavy use of another type known as `Bottom`, which can also be written as `Union{}`. This corresponds to the empty set.

Julia's types support the standard operations of set theory: you can ask whether T1 is a "subset" (subtype) of T2 with T1 `<:` T2. Likewise, you intersect two types using `typeintersect`, take their union with `Union`, and compute a type that contains their union with `typejoin`:

```
julia> typeintersect(Int, Float64)
Union{}

julia> Union{Int, Float64}
Union{Float64, Int64}

julia> typejoin(Int, Float64)
Real

julia> typeintersect(Signed, Union{UInt8, Int8})
Int8

julia> Union{Signed, Union{UInt8, Int8}}
Union{UInt8, Signed}

julia> typejoin(Signed, Union{UInt8, Int8})
Integer

julia> typeintersect(Tuple{Integer, Float64}, Tuple{Int, Real})
Tuple{Int64, Float64}

julia> Union{Tuple{Integer, Float64}, Tuple{Int, Real}}
Union{Tuple{Int64, Real}, Tuple{Integer, Float64}}

julia> typejoin(Tuple{Integer, Float64}, Tuple{Int, Real})
Tuple{Integer, Real}
```

While these operations may seem abstract, they lie at the heart of Julia. For example, method dispatch is implemented by stepping through the items in a method list until reaching one for which the type of the argument tuple is a subtype of the method signature. For this algorithm to work, it's important that methods be sorted by their specificity, and that the search begins with the most specific methods. Consequently, Julia also implements a partial order on types; this is achieved by functionality that is similar to `<:`, but with differences that will be discussed below.

UnionAll types

Julia's type system can also express an *iterated union* of types: a union of types over all values of some variable. This is needed to describe parametric types where the values of some parameters are not known.

For example, `Array` has two parameters as in `Array{Int,2}`. If we did not know the element type, we could write `Array{T,2} where T`, which is the union of `Array{T,2}` for all values of `T`: `Union{Array{Int8,2}, Array{Int16,2}, ...}`.

Such a type is represented by a `UnionAll` object, which contains a variable (`T` in this example, of type `TypeVar`), and a wrapped type (`Array{T,2}` in this example).

Consider the following methods:

```
f1(A::Array) = 1
f2(A::Array{Int}) = 2
f3(A::Array{T}) where {T<:Any} = 3
f4(A::Array{Any}) = 4
```

The signature - as described in Function calls - of `f3` is a `UnionAll` type wrapping a tuple type: `Tuple{typeof(f3), Array{T}} where T`. All but `f4` can be called with `a = [1,2]`; all but `f2` can be called with `b = Any[1,2]`.

Let's look at these types a little more closely:

```
julia> dump(Array)
UnionAll
  var: TypeVar
    name: Symbol T
    lb: Union{}
    ub: abstract type Any
  body: UnionAll
    var: TypeVar
      name: Symbol N
      lb: Union{}
      ub: abstract type Any
    body: mutable struct Array{T, N} <: DenseArray{T, N}
      ref::MemoryRef{T}
      size::NTuple{N, Int64}
```

This indicates that `Array` actually names a `UnionAll` type. There is one `UnionAll` type for each parameter, nested. The syntax `Array{Int,2}` is equivalent to `Array{Int}{2}`; internally each `UnionAll` is instantiated with a particular variable value, one at a time, outermost-first. This gives a natural meaning to the omission of trailing type parameters; `Array{Int}` gives a type equivalent to `Array{Int,N} where N`.

A `TypeVar` is not itself a type, but rather should be considered part of the structure of a `UnionAll` type. Type variables have lower and upper bounds on their values (in the fields `lb` and `ub`). The symbol `name` is purely cosmetic. Internally, `TypeVars` are compared by address, so they are defined as mutable types to ensure that "different" type variables can be distinguished. However, by convention they should not be mutated.

One can construct `TypeVars` manually:

```
julia> TypeVar(:V, Signed, Real)
Signed<:V<:Real
```

There are convenience versions that allow you to omit any of these arguments except the `name` symbol.

The syntax `Array{T} where T<:Integer` is lowered to

```
let T = TypeVar(:T,Integer)
    UnionAll(T, Array{T})
end
```

so it is seldom necessary to construct a `TypeVar` manually (indeed, this is to be avoided).

Free variables

The concept of a *free* type variable is extremely important in the type system. We say that a variable `V` is free in type `T` if `T` does not contain the `UnionAll` that introduces variable `V`. For example, the type `Array{Array{V} where V<:Integer}` has no free variables, but the `Array{V}` part inside of it does have a free variable, `V`.

A type with free variables is, in some sense, not really a type at all. Consider the type `Array{Array{T}} where T`, which refers to all homogeneous arrays of arrays. The inner type `Array{T}`, seen by itself, might seem to refer to any kind of array. However, every element of the outer array must have the *same* array type, so `Array{T}` cannot refer to just any old array. One could say that `Array{T}` effectively "occurs" multiple times, and `T` must have the same value each "time".

For this reason, the function `jl_has_free_typevars` in the C API is very important. Types for which it returns true will not give meaningful answers in subtyping and other type functions.

TypeNames

The following two `Array` types are functionally equivalent, yet print differently:

```
julia> TV, NV = TypeVar(:T), TypeVar(:N)
(T, N)

julia> Array
Array

julia> Array{TV, NV}
Array{T, N}
```

These can be distinguished by examining the `name` field of the type, which is an object of type `TypeName`:

```
julia> dump(Array{Int,1}.name)
TypeName
  name: Symbol Array
  module: Module Core
  singletonname: Symbol Array
  names: SimpleVector
    1: Symbol ref
    2: Symbol size
  atomicfields: Ptr{Nothing}(0x0000000000000000)
  constfields: Ptr{Nothing}(0x0000000000000000)
```

```
  wrapper: UnionAll
    var: TypeVar
      name: Symbol T
      lb: Union{}
      ub: abstract type Any
    body: UnionAll
      var: TypeVar
        name: Symbol N
        lb: Union{}
        ub: abstract type Any
      body: mutable struct Array{T, N} <: DenseArray{T, N}
  Typeofwrapper: abstract type Type{Array} <: Any
  cache: SimpleVector
    ...
  linearcache: SimpleVector
    ...
  hash: Int64 2594190783455944385
  backedges: #undef
  partial: #undef
  max_args: Int32 0
  n_uninitialized: Int32 0
  flags: UInt8 0x02
  cache_entry_count: UInt8 0x00
  max_methods: UInt8 0x00
  constprop_heuristic: UInt8 0x00
```

In this case, the relevant field is `wrapper`, which holds a reference to the top-level type used to make new `Array` types.

```
julia> pointer_from_objref(Array)
Ptr{Cvoid} @0x00007fcc7de64850

julia> pointer_from_objref(Array.body.body.name.wrapper)
Ptr{Cvoid} @0x00007fcc7de64850

julia> pointer_from_objref(Array{TV,NV})
Ptr{Cvoid} @0x00007fcc80c4d930

julia> pointer_from_objref(Array{TV,NV}.name.wrapper)
Ptr{Cvoid} @0x00007fcc7de64850
```

The `wrapper` field of `Array` points to itself, but for `Array{TV,NV}` it points back to the original definition of the type.

What about the other fields? `hash` assigns an integer to each type. To examine the `cache` field, it's helpful to pick a type that is less heavily used than Array. Let's first create our own type:

```
julia> struct MyType{T,N} end

julia> MyType{Int,2}
MyType{Int64, 2}
```

```
julia> MyType{Float32, 5}
MyType{Float32, 5}
```

When you instantiate a parametric type, each concrete type gets saved in a type cache (`MyType.body.body.name.cache`). However, instances containing free type variables are not cached.

Tuple types

Tuple types constitute an interesting special case. For dispatch to work on declarations like `x::Tuple`, the type has to be able to accommodate any tuple. Let's check the parameters:

```
julia> Tuple
Tuple

julia> Tuple.parameters
svec(Vararg{Any})
```

Unlike other types, tuple types are covariant in their parameters, so this definition permits `Tuple` to match any type of tuple:

```
julia> typeintersect(Tuple, Tuple{Int,Float64})
Tuple{Int64, Float64}

julia> typeintersect(Tuple{Vararg{Any}}, Tuple{Int,Float64})
Tuple{Int64, Float64}
```

However, if a variadic (`Vararg`) tuple type has free variables it can describe different kinds of tuples:

```
julia> typeintersect(Tuple{Vararg{T} where T}, Tuple{Int,Float64})
Tuple{Int64, Float64}

julia> typeintersect(Tuple{Vararg{T}} where T, Tuple{Int,Float64})
Union{}
```

Notice that when T is free with respect to the `Tuple` type (i.e. its binding `UnionAll` type is outside the `Tuple` type), only one T value must work over the whole type. Therefore a heterogeneous tuple does not match.

Finally, it's worth noting that `Tuple{}` is distinct:

```
julia> Tuple{}
Tuple{}

julia> Tuple{}.parameters
svec()

julia> typeintersect(Tuple{}, Tuple{Int})
Union{}
```

What is the "primary" tuple-type?

```
julia> pointer_from_objref(Tuple)
Ptr{Cvoid} @0x00007f5998a04370

julia> pointer_from_objref(Tuple{})
Ptr{Cvoid} @0x00007f5998a570d0

julia> pointer_from_objref(Tuple.name.wrapper)
Ptr{Cvoid} @0x00007f5998a04370

julia> pointer_from_objref(Tuple{}.name.wrapper)
Ptr{Cvoid} @0x00007f5998a04370
```

so `Tuple == Tuple{Vararg{Any}}` is indeed the primary type.

Diagonal types

Consider the type `Tuple{T,T} where T`. A method with this signature would look like:

```
f(x::T, y::T) where {T} = ...
```

According to the usual interpretation of a `UnionAll` type, this T ranges over all types, including `Any`, so this type should be equivalent to `Tuple{Any,Any}`. However, this interpretation causes some practical problems.

First, a value of T needs to be available inside the method definition. For a call like `f(1, 1.0)`, it's not clear what T should be. It could be `Union{Int,Float64}`, or perhaps `Real`. Intuitively, we expect the declaration `x::T` to mean `T === typeof(x)`. To make sure that invariant holds, we need `typeof(x) === typeof(y) === T` in this method. That implies the method should only be called for arguments of the exact same type.

It turns out that being able to dispatch on whether two values have the same type is very useful (this is used by the promotion system for example), so we have multiple reasons to want a different interpretation of `Tuple{T,T} where T`. To make this work we add the following rule to subtyping: if a variable occurs more than once in covariant position, it is restricted to ranging over only concrete types. ("Covariant position" means that only `Tuple` and `Union` types occur between an occurrence of a variable and the `UnionAll` type that introduces it.) Such variables are called "diagonal variables" or "concrete variables".

So for example, `Tuple{T,T} where T` can be seen as `Union{Tuple{Int8,Int8}, Tuple{Int16,Int16}, ...}`, where T ranges over all concrete types. This gives rise to some interesting subtyping results. For example `Tuple{Real,Real}` is not a subtype of `Tuple{T,T} where T`, because it includes some types like `Tuple{Int8,Int16}` where the two elements have different types. `Tuple{Real,Real}` and `Tuple{T,T} where T` have the non-trivial intersection `Tuple{T,T} where T<:Real`. However, `Tuple{Real}` *is* a subtype of `Tuple{T} where T`, because in that case T occurs only once and so is not diagonal.

Next consider a signature like the following:

```
f(a::Array{T}, x::T, y::T) where {T} = ...
```

In this case, T occurs in invariant position inside `Array{T}`. That means whatever type of array is passed unambiguously determines the value of T – we say T has an *equality constraint* on it. Therefore in this case the diagonal rule is not really necessary, since the array determines T and we can then allow x and y to be of any subtypes of T. So variables that occur in invariant position are never considered diagonal. This choice of behavior is slightly controversial – some feel this definition should be written as

```
f(a::Array{T}, x::S, y::S) where {T, S<:T} = ...
```

to clarify whether x and y need to have the same type. In this version of the signature they would, or we could introduce a third variable for the type of y if x and y can have different types.

The next complication is the interaction of unions and diagonal variables, e.g.

```
f(x::Union{Nothing,T}, y::T) where {T} = ...
```

Consider what this declaration means. y has type T. x then can have either the same type T, or else be of type `Nothing`. So all of the following calls should match:

```
f(1, 1)
f("", "")
f(2.0, 2.0)
f(nothing, 1)
f(nothing, "")
f(nothing, 2.0)
```

These examples are telling us something: when x is `nothing::Nothing`, there are no extra constraints on y. It is as if the method signature had `y::Any`. Indeed, we have the following type equivalence:

```
(Tuple{Union{Nothing,T},T} where T) == Union{Tuple{Nothing,Any}, Tuple{T,T} where T}
```

The general rule is: a concrete variable in covariant position acts like it's not concrete if the subtyping algorithm only *uses* it once. When x has type `Nothing`, we don't need to use the T in `Union{Nothing,T}`; we only use it in the second slot. This arises naturally from the observation that in `Tuple{T} where T` restricting T to concrete types makes no difference; the type is equal to `Tuple{Any}` either way.

However, appearing in *invariant* position disqualifies a variable from being concrete whether that appearance of the variable is used or not. Otherwise types can behave differently depending on which other types they are compared to, making subtyping not transitive. For example, consider

```
Tuple{Int,Int8,Vector{Integer}} <: Tuple{T,T,Vector{Union{Integer,T}}} where T
```

If the T inside the `Union` is ignored, then T is concrete and the answer is "false" since the first two types aren't the same. But consider instead

```
Tuple{Int,Int8,Vector{Any}} <: Tuple{T,T,Vector{Union{Integer,T}}} where T
```

Now we cannot ignore the T in the `Union` (we must have `T == Any`), so T is not concrete and the answer is "true". That would make the concreteness of T depend on the other type, which is not acceptable since a type must have a clear meaning on its own. Therefore the appearance of T inside `Vector` is considered in both cases.

Subtyping diagonal variables

The subtyping algorithm for diagonal variables has two components: (1) identifying variable occurrences, and (2) ensuring that diagonal variables range over concrete types only.

The first task is accomplished by keeping counters `occurs_inv` and `occurs_cov` (in `src/subtype.c`) for each variable in the environment, tracking the number of invariant and covariant occurrences, respectively. A variable is diagonal when `occurs_inv == 0 && occurs_cov > 1`.

The second task is accomplished by imposing a condition on a variable's lower bound. As the subtyping algorithm runs, it narrows the bounds of each variable (raising lower bounds and lowering upper bounds) to keep track of the range of variable values for which the subtype relation would hold. When we are done evaluating the body of a `UnionAll` type whose variable is diagonal, we look at the final values of the bounds. Since the variable must be concrete, a contradiction occurs if its lower bound could not be a subtype of a concrete type. For example, an abstract type like `AbstractArray` cannot be a subtype of a concrete type, but a concrete type like `Int` can be, and the empty type `Bottom` can be as well. If a lower bound fails this test the algorithm stops with the answer `false`.

For example, in the problem `Tuple{Int,String} <: Tuple{T,T} where T`, we derive that this would be true if `T` were a supertype of `Union{Int,String}`. However, `Union{Int,String}` is an abstract type, so the relation does not hold.

This concreteness test is done by the function `is_leaf_bound`. Note that this test is slightly different from `jl_is_leaf_type`, since it also returns `true` for `Bottom`. Currently this function is heuristic, and does not catch all possible concrete types. The difficulty is that whether a lower bound is concrete might depend on the values of other type variable bounds. For example, `Vector{T}` is equivalent to the concrete type `Vector{Int}` only if both the upper and lower bounds of `T` equal `Int`. We have not yet worked out a complete algorithm for this.

Introduction to the internal machinery

Most operations for dealing with types are found in the files `jltypes.c` and `subtype.c`. A good way to start is to watch subtyping in action. Build Julia with `make debug` and fire up Julia within a debugger. gdb debugging tips has some tips which may be useful.

Because the subtyping code is used heavily in the REPL itself – and hence breakpoints in this code get triggered often – it will be easiest if you make the following definition:

```
julia> function mysubtype(a,b)
           ccall(:jl_breakpoint, Cvoid, (Any,), nothing)
           a <: b
       end
```

and then set a breakpoint in `jl_breakpoint`. Once this breakpoint gets triggered, you can set breakpoints in other functions.

As a warm-up, try the following:

```
mysubtype(Tuple{Int, Float64}, Tuple{Integer, Real})
```

We can make it more interesting by trying a more complex case:

```
mysubtype(Tuple{Array{Int,2}, Int8}, Tuple{Array{T}, T} where T)
```

Subtyping and method sorting

The `type_morespecific` functions are used for imposing a partial order on functions in method tables (from most-to-least specific). Specificity is strict; if a is more specific than b, then a does not equal b and b is not more specific than a.

If a is a strict subtype of b, then it is automatically considered more specific. From there, `type_morespecific` employs some less formal rules. For example, `subtype` is sensitive to the number of arguments, but `type_morespecific` may not be. In particular, `Tuple{Int,AbstractFloat}` is more specific than `Tuple{Integer}`, even though it is not a subtype. (Of `Tuple{Int,AbstractFloat}` and `Tuple{Integer,Float64}`, neither is more specific than the other.) Likewise, `Tuple{Int,Vararg{Int}}` is not a subtype of `Tuple{Integer}`, but it is considered more specific. However, `morespecific` does get a bonus for length: in particular, `Tuple{Int,Int}` is more specific than `Tuple{Int,Vararg{Int}}`.

Additionally, if 2 methods are defined with identical signatures, per type-equal, then they will instead be compared by order of addition, such that the later method is more specific than the earlier one.

108.4 Memory layout of Julia Objects

Object layout (`jl_value_t`)

The `jl_value_t` struct is the name for a block of memory owned by the Julia Garbage Collector, representing the data associated with a Julia object in memory. Absent any type information, it is simply an opaque pointer:

```
typedef struct jl_value_t* jl_pvalue_t;
```

Each `jl_value_t` struct is contained in a `jl_typetag_t` struct that contains metadata information about the Julia object, such as its type and garbage collector (gc) reachability:

```
typedef struct {
    opaque metadata;
    jl_value_t value;
} jl_typetag_t;
```

The type of any Julia object is an instance of a leaf `jl_datatype_t` object. The `jl_typeof()` function can be used to query for it:

```
jl_value_t *jl_typeof(jl_value_t *v);
```

The layout of the object depends on its type. Reflection methods can be used to inspect that layout. A field can be accessed by calling one of the get-field methods:

```
jl_value_t *jl_get_nth_field_checked(jl_value_t *v, size_t i);
jl_value_t *jl_get_field(jl_value_t *o, char *fld);
```

If the field types are known, a priori, to be all pointers, the values can also be extracted directly as an array access:

```
jl_value_t *v = value->fieldptr[n];
```

As an example, a "boxed" `uint16_t` is stored as follows:

```
struct {
    opaque metadata;
    struct {
        uint16_t data;        // -- 2 bytes
    } jl_value_t;
};
```

This object is created by `jl_box_uint16()`. Note that the `jl_value_t` pointer references the data portion, not the metadata at the top of the struct.

A value may be stored "unboxed" in many circumstances (just the data, without the metadata, and possibly not even stored but just kept in registers), so it is unsafe to assume that the address of a box is a unique identifier. The "egal" test (corresponding to the === function in Julia), should instead be used to compare two unknown objects for equivalence:

```
int jl_egal(jl_value_t *a, jl_value_t *b);
```

This optimization should be relatively transparent to the API, since the object will be "boxed" on-demand, whenever a `jl_value_t` pointer is needed.

Note that modification of a `jl_value_t` pointer in memory is permitted only if the object is mutable. Otherwise, modification of the value may corrupt the program and the result will be undefined. The mutability property of a value can be queried for with:

```
int jl_is_mutable(jl_value_t *v);
```

If the object being stored is a `jl_value_t`, the Julia garbage collector must be notified also:

```
void jl_gc_wb(jl_value_t *parent, jl_value_t *ptr);
```

However, the Embedding Julia section of the manual is also required reading at this point, for covering other details of boxing and unboxing various types, and understanding the gc interactions.

Mirror structs for some of the built-in types are defined in `julia.h`. The corresponding global `jl_datatype_t` objects are created by `jl_init_types` in `jltypes.c`.

Garbage collector mark bits

The garbage collector uses several bits from the metadata portion of the `jl_typetag_t` to track each object in the system. Further details about this algorithm can be found in the comments of the garbage collector implementation in `gc-stock.c`.

Object allocation

Most new objects are allocated by `jl_new_structv()`:

```
jl_value_t *jl_new_struct(jl_datatype_t *type, ...);
jl_value_t *jl_new_structv(jl_datatype_t *type, jl_value_t **args, uint32_t na);
```

Although, `isbits` objects can be also constructed directly from memory:

```
jl_value_t *jl_new_bits(jl_value_t *bt, void *data)
```

And some objects have special constructors that must be used instead of the above functions:

Types:

```
jl_datatype_t *jl_apply_type(jl_datatype_t *tc, jl_tuple_t *params);
jl_datatype_t *jl_apply_array_type(jl_datatype_t *type, size_t dim);
```

While these are the most commonly used options, there are more low-level constructors too, which you can find declared in `julia.h`. These are used in `jl_init_types()` to create the initial types needed to bootstrap the creation of the Julia system image.

Tuples:

```
jl_tuple_t *jl_tuple(size_t n, ...);
jl_tuple_t *jl_tuplev(size_t n, jl_value_t **v);
jl_tuple_t *jl_alloc_tuple(size_t n);
```

The representation of tuples is highly unique in the Julia object representation ecosystem. In some cases, a `Base.tuple()` object may be an array of pointers to the objects contained by the tuple equivalent to:

```
typedef struct {
    size_t length;
    jl_value_t *data[length];
} jl_tuple_t;
```

However, in other cases, the tuple may be converted to an anonymous `isbits` type and stored unboxed, or it may not stored at all (if it is not being used in a generic context as a `jl_value_t*`).

Symbols:

```
jl_sym_t *jl_symbol(const char *str);
```

Functions and MethodInstance:

```
jl_value_t *jl_new_generic_function(jl_sym_t *name);
jl_method_instance_t *jl_new_method_instance(jl_value_t *ast, jl_tuple_t *sparams);
```

Arrays:

```
jl_array_t *jl_new_array(jl_value_t *atype, jl_tuple_t *dims);
jl_array_t *jl_alloc_array_1d(jl_value_t *atype, size_t nr);
jl_array_t *jl_alloc_array_nd(jl_value_t *atype, size_t *dims, size_t ndims);
```

Note that many of these have alternative allocation functions for various special-purposes. The list here reflects the more common usages, but a more complete list can be found by reading the `julia.h` header file.

Internal to Julia, storage is typically allocated by `newstruct()` (or `newobj()` for the special types):

```
jl_value_t *newstruct(jl_value_t *type);
jl_value_t *newobj(jl_value_t *type, size_t nfields);
```

And at the lowest level, memory is getting allocated by a call to the garbage collector (in `gc-stock.c`), then tagged with its type:

```
jl_value_t *jl_gc_allocobj(size_t nbytes);
void jl_set_typeof(jl_value_t *v, jl_datatype_t *type);
```

Out of date Warning

The documentation and usage for the function `jl_gc_allocobj` may be out of date

Note that all objects are allocated in multiples of 4 bytes and aligned to the platform pointer size. Memory is allocated from a pool for smaller objects, or directly with `malloc()` for large objects.

Singleton Types

Singleton types have only one instance and no data fields. Singleton instances have a size of 0 bytes, and consist only of their metadata. e.g. `nothing::Nothing`.

See Singleton Types and Nothingness and missing values

108.5 Eval of Julia code

One of the hardest parts about learning how the Julia Language runs code is learning how all of the pieces work together to execute a block of code.

Each chunk of code typically makes a trip through many steps with potentially unfamiliar names, such as (in no particular order): flisp, AST, C++, LLVM, `eval`, `typeinf`, `macroexpand`, sysimg (or system image), bootstrapping, compile, parse, execute, JIT, interpret, box, unbox, intrinsic function, and primitive function, before turning into the desired result (hopefully).

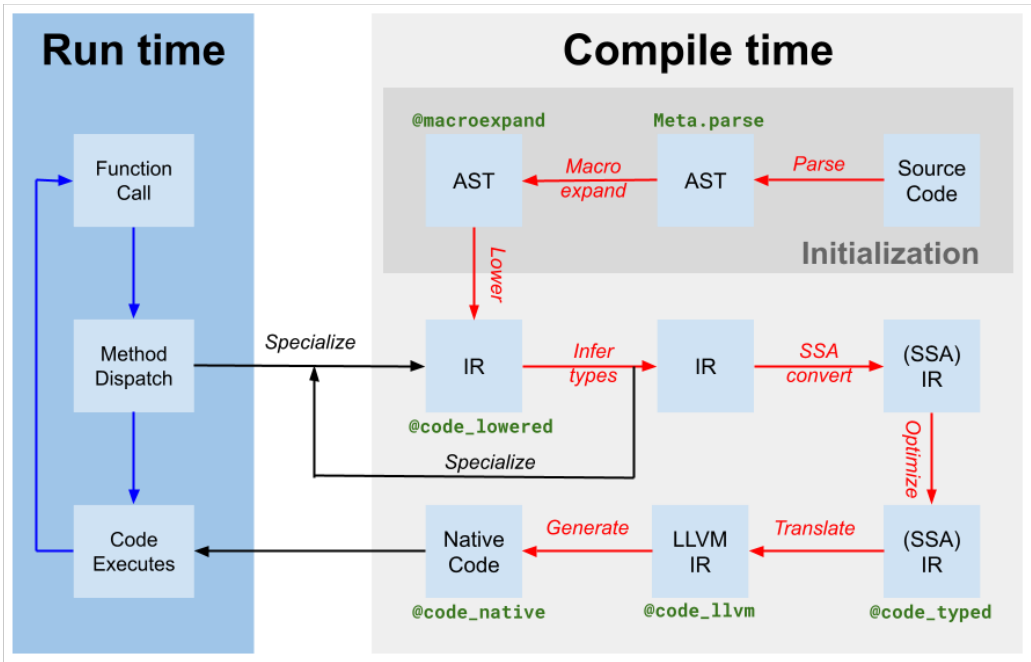


Figure 108.1: Diagram of the compiler flow

Definitions

- REPL

  REPL stands for Read-Eval-Print Loop. It's just what we call the command line environment for short.

- AST

  Abstract Syntax Tree. The AST is the digital representation of the code structure. In this form the code has been tokenized for meaning so that it is more suitable for manipulation and execution.

Julia Execution

The 10,000 foot view of the whole process is as follows:

1. The user starts `julia`.
2. The C function `main()` from `cli/loader_exe.c` gets called. This function processes the command line arguments, filling in the `jl_options` struct and setting the variable ARGS. It then initializes Julia

(by calling `julia_init` in `init.c`, which may load a previously compiled sysimg). Finally, it passes off control to Julia by calling `Base._start()`.

3. When `_start()` takes over control, the subsequent sequence of commands depends on the command line arguments given. For example, if a filename was supplied, it will proceed to execute that file. Otherwise, it will start an interactive REPL.

4. Skipping the details about how the REPL interacts with the user, let's just say the program ends up with a block of code that it wants to run.

5. If the block of code to run is in a file, `jl_load(char *filename)` gets invoked to load the file and parse it. Each fragment of code is then passed to `eval` to execute.

6. Each fragment of code (or AST), is handed off to `eval()` to turn into results.

7. `eval()` takes each code fragment and tries to run it in `jl_toplevel_eval_flex()`.

8. `jl_toplevel_eval_flex()` decides whether the code is a "toplevel" action (such as `using` or `module`), which would be invalid inside a function. If so, it passes off the code to the toplevel interpreter.

9. `jl_toplevel_eval_flex()` then expands the code to eliminate any macros and to "lower" the AST to make it simpler to execute.

10. `jl_toplevel_eval_flex()` then uses some simple heuristics to decide whether to JIT compile the AST or to interpret it directly.

11. The bulk of the work to interpret code is handled by `eval` in `interpreter.c`.

12. If instead, the code is compiled, the bulk of the work is handled by `codegen.cpp`. Whenever a Julia function is called for the first time with a given set of argument types, type inference will be run on that function. This information is used by the codegen step to generate faster code.

13. Eventually, the user quits the REPL, or the end of the program is reached, and the `_start()` method returns.

14. Just before exiting, `main()` calls `jl_atexit_hook(exit_code)`. This calls `Base._atexit()` (which calls any functions registered to `atexit()` inside Julia). Then it calls `jl_gc_run_all_finalizers()`. Finally, it gracefully cleans up all `libuv` handles and waits for them to flush and close.

Parsing

By default, Julia uses JuliaSyntax.jl to produce the AST. Historically, it used a small lisp program written in femtolisp, the source-code for which is distributed inside Julia in src/flisp. If the `JULIA_USE_FLISP_PARSER` environment variable is set to 1, the old parser will be used instead.

Macro Expansion

When `eval()` encounters a macro, it expands that AST node before attempting to evaluate the expression. Macro expansion involves a handoff from `eval()` (in Julia), to the parser function `jl_macroexpand()` (written in `flisp`) to the Julia macro itself (written in - what else - Julia) via `fl_invoke_julia_macro()`, and back.

Typically, macro expansion is invoked as a first step during a call to `Meta.lower()`/`Core._lower()`, although it can also be invoked directly by a call to `macroexpand()`/`jl_macroexpand()`.

Type Inference

Type inference is implemented in Julia by `typeinf()` in `compiler/typeinfer.jl`. Type inference is the process of examining a Julia function and determining bounds for the types of each of its variables, as well as bounds on the type of the return value from the function. This enables many future optimizations, such as unboxing of known immutable values, and compile-time hoisting of various run-time operations such as computing field offsets and function pointers. Type inference may also include other steps such as constant propagation and inlining.

More Definitions

- JIT

  Just-In-Time Compilation The process of generating native-machine code into memory right when it is needed.

- LLVM

  Low-Level Virtual Machine (a compiler) The Julia JIT compiler is a program/library called libLLVM. Codegen in Julia refers both to the process of taking a Julia AST and turning it into LLVM instructions, and the process of LLVM optimizing that and turning it into native assembly instructions.

- C++

  The programming language that LLVM is implemented in, which means that codegen is also implemented in this language. The rest of Julia's library is implemented in C, in part because its smaller feature set makes it more usable as a cross-language interface layer.

- box

  This term is used to describe the process of taking a value and allocating a wrapper around the data that is tracked by the garbage collector (gc) and is tagged with the object's type.

- unbox

  The reverse of boxing a value. This operation enables more efficient manipulation of data when the type of that data is fully known at compile-time (through type inference).

- generic function

  A Julia function composed of multiple "methods" that are selected for dynamic dispatch based on the argument type-signature

- anonymous function or "method"

  A Julia function without a name and without type-dispatch capabilities

- primitive function

  A function implemented in C but exposed in Julia as a named function "method" (albeit without generic function dispatch capabilities, similar to a anonymous function)

- intrinsic function

  A low-level operation exposed as a function in Julia. These pseudo-functions implement operations on raw bits such as add and sign extend that cannot be expressed directly in any other way. Since they operate on bits directly, they must be compiled into a function and surrounded by a call to `Core.Intrinsics.box(T, ...)` to reassign type information to the value.

JIT Code Generation

Codegen is the process of turning a Julia AST into native machine code.

The JIT environment is initialized by an early call to `jl_init_codegen` in `codegen.cpp`.

On demand, a Julia method is converted into a native function by the function `emit_function(jl_method_instance_t*)`.

(note, when using the MCJIT (in LLVM v3.4+), each function must be JIT into a new module.) This function recursively calls `emit_expr()` until the entire function has been emitted.

Much of the remaining bulk of this file is devoted to various manual optimizations of specific code patterns. For example, `emit_known_call()` knows how to inline many of the primitive functions (defined in `builtins.c`) for various combinations of argument types.

Other parts of codegen are handled by various helper files:

- `debuginfo.cpp`

  Handles backtraces for JIT functions

- `ccall.cpp`

  Handles the ccall and llvmcall FFI, along with various `abi_*.cpp` files

- `intrinsics.cpp`

  Handles the emission of various low-level intrinsic functions

Bootstrapping

The process of creating a new system image is called "bootstrapping".

The etymology of this word comes from the phrase "pulling oneself up by the bootstraps", and refers to the idea of starting from a very limited set of available functions and definitions and ending with the creation of a full-featured environment.

System Image

The system image is a precompiled archive of a set of Julia files. The `sys.ji` file distributed with Julia is one such system image, generated by executing the file `sysimg.jl`, and serializing the resulting environment (including Types, Functions, Modules, and all other defined values) into a file. Therefore, it contains a frozen version of the `Main`, `Core`, and `Base` modules (and whatever else was in the environment at the end of bootstrapping). This serializer/deserializer is implemented by `jl_save_system_image/jl_restore_system_image in staticdata.c`.

If there is no sysimg file (`jl_options.image_file == NULL`), this also implies that `--build` was given on the command line, so the final result should be a new sysimg file. During Julia initialization, minimal `Core` and `Main` modules are created. Then a file named `boot.jl` is evaluated from the current directory. Julia then evaluates any file given as a command line argument until it reaches the end. Finally, it saves the resulting environment to a "sysimg" file for use as a starting point for a future Julia run.

108.6 Calling Conventions

Julia uses three calling conventions for four distinct purposes:

Julia Native Calling Convention

The native calling convention is designed for fast non-generic calls. It usually uses a specialized signature.

- LLVM ghosts (zero-length types) are omitted.

Name	Prefix	Purpose
Native	`julia_`	Speed via specialized signatures
JL Call	`jlcall_`	Wrapper for generic calls
JL Call	`jl_`	Builtins
C ABI	`jlcapi_`	Wrapper callable from C

- LLVM scalars and vectors are passed by value.
- LLVM aggregates (arrays and structs) are passed by reference.

A small return value is returned as LLVM return values. A large return value is returned via the "structure return" (`sret`) convention, where the caller provides a pointer to a return slot.

An argument or return value that is a homogeneous tuple is sometimes represented as an LLVM vector instead of an LLVM array.

JL Call Convention

The JL Call convention is for builtins and generic dispatch. Hand-written functions using this convention are declared via the macro `JL_CALLABLE`. The convention uses exactly 3 parameters:

- F - Julia representation of function that is being applied
- `args` - pointer to array of pointers to boxes
- `nargs` - length of the array

The return value is a pointer to a box.

C ABI

C ABI wrappers enable calling Julia from C. The wrapper calls a function using the native calling convention.

Tuples are always represented as C arrays.

108.7 High-level Overview of the Native-Code Generation Process

Representation of Pointers

When emitting code to an object file, pointers will be emitted as relocations. The deserialization code will ensure any object that pointed to one of these constants gets recreated and contains the right runtime pointer.

Otherwise, they will be emitted as literal constants.

To emit one of these objects, call `literal_pointer_val`. It'll handle tracking the Julia value and the LLVM global, ensuring they are valid both for the current runtime and after deserialization.

When emitted into the object file, these globals are stored as references in a large `gvals` table. This allows the deserializer to reference them by index, and implement a custom manual mechanism similar to a Global Offset Table (GOT) to restore them.

Function pointers are handled similarly. They are stored as values in a large `fvals` table. Like globals, this allows the deserializer to reference them by index.

Note that `extern` functions are handled separately, with names, via the usual symbol resolution mechanism in the linker.

Note too that `ccall` functions are also handled separately, via a manual GOT and Procedure Linkage Table (PLT).

Representation of Intermediate Values

Values are passed around in a `jl_cgval_t` struct. This represents an R-value, and includes enough information to determine how to assign or pass it somewhere.

They are created via one of the helper constructors, usually: `mark_julia_type` (for immediate values) and `mark_julia_slot` (for pointers to values).

The function `convert_julia_type` can transform between any two types. It returns an R-value with `cgval.typ` set to `typ`. It'll cast the object to the requested representation, making heap boxes, allocating stack copies, and computing tagged unions as needed to change the representation.

By contrast `update_julia_type` will change `cgval.typ` to `typ`, only if it can be done at zero-cost (i.e. without emitting any code).

Union representation

Inferred union types may be stack allocated via a tagged type representation.

The primitive routines that need to be able to handle tagged unions are:

- mark-type
- load-local
- store-local
- isa
- is
- emit_typeof
- emit_sizeof
- boxed
- unbox
- specialized cc-ret

Everything else should be possible to handle in inference by using these primitives to implement union-splitting.

The representation of the tagged-union is as a pair of `< void* union, byte selector >`. The selector is fixed-size as `byte & 0x7f`, and will union-tag the first 126 isbits. It records the one-based depth-first count into the type-union of the isbits objects inside. An index of zero indicates that the `union*` is actually

a tagged heap-allocated `jl_value_t*`, and needs to be treated as normal for a boxed object rather than as a tagged union.

The high bit of the selector (`byte & 0x80`) can be tested to determine if the `void*` is actually a heap-allocated (`jl_value_t*`) box, thus avoiding the cost of re-allocating a box, while maintaining the ability to efficiently handle union-splitting based on the low bits.

It is guaranteed that `byte & 0x7f` is an exact test for the type, if the value can be represented by a tag – it will never be marked `byte = 0x80`. It is not necessary to also test the type-tag when testing `isa`.

The `union*` memory region may be allocated at *any* size. The only constraint is that it is big enough to contain the data currently specified by `selector`. It might not be big enough to contain the union of all types that could be stored there according to the associated Union type field. Use appropriate care when copying.

Specialized Calling Convention Signature Representation

A `jl_returninfo_t` object describes the specialized calling convention details of any callable. It can be generated from any (specTypes, rettype) pair, such as a CodeInstance, or other place they are declared. This is the expected calling convention for specptr, but other data may be stored there. Only if the function pointer stored there has the expected specialized calling convention will the corresponding flag be set in specsigflags to indicate it is useable.

If any of the arguments or return type of a method can be represented unboxed, and none are unable to be represented unboxed (such as an unbounded vararg), it will be given an optimized calling convention signature based on the `specTypes` and `rettype` values.

The general principles are that:

- Primitive types get passed in int/float registers.
- Tuples of VecElement types get passed in vector registers.
- Structs get passed on the stack.
- Return values are handled similarly to arguments, with a size-cutoff at which they will instead be returned via a hidden sret argument.

The total logic for this is implemented by `get_specsig_function` and `deserves_sret`.

Additionally, if the return type is a union, it may be returned as a pair of values (a pointer and a tag). If the union values can be stack-allocated, then sufficient space to store them will also be passed as a hidden first argument. If the struct to return needs gc roots, space for those will be passed as a hidden second argument. It is up to the callee whether the returned pointer will point to this space, a boxed object, or even other constant memory.

108.8 Julia Functions

This document will explain how functions, method definitions, and method tables work.

Method Tables

Every function in Julia is a generic function. A generic function is conceptually a single function, but consists of many definitions, or methods. The methods of a generic function are stored in a method table. There is one global method table (type `MethodTable`) named `Core.methodtable`. Any default operation on methods (such as calls) uses that table.

Function calls

Given the call `f(x, y)`, the following steps are performed: First, a tuple type is formed, `Tuple{typeof(f), typeof(x), typeof(y)}`. Note that the type of the function itself is the first element. This is because the function itself participates symmetrically in method lookup with the other arguments. This tuple type is looked up in the global method table. However, the system can then cache the results, so these steps can be skipped later for similar lookups.

This dispatch process is performed by `jl_apply_generic`, which takes two arguments: a pointer to an array of the values `f`, `x`, and `y`, and the number of values (in this case 3).

Throughout the system, there are two kinds of APIs that handle functions and argument lists: those that accept the function and arguments separately, and those that accept a single argument structure. In the first kind of API, the "arguments" part does *not* contain information about the function, since that is passed separately. In the second kind of API, the function is the first element of the argument structure.

For example, the following function for performing a call accepts just an `args` pointer, so the first element of the args array will be the function to call:

```
jl_value_t *jl_apply(jl_value_t **args, uint32_t nargs)
```

This entry point for the same functionality accepts the function separately, so the `args` array does not contain the function:

```
jl_value_t *jl_call(jl_value_t *f, jl_value_t **args, int32_t nargs);
```

Adding methods

Given the above dispatch process, conceptually all that is needed to add a new method is (1) a tuple type, and (2) code for the body of the method. `jl_method_def` implements this operation.

Creating generic functions

Since every object is callable, nothing special is needed to create a generic function. Therefore `jl_new_generic_function` simply creates a new singleton (0 size) subtype of `Function` and returns its instance. A function can have a mnemonic "display name" which is used in debug info and when printing objects. For example the name of `Base.sin` is `sin`. By convention, the name of the created *type* is the same as the function name, with a `#` prepended. So `typeof(sin)` is `Base.#sin`.

Closures

A closure is simply a callable object with field names corresponding to captured variables. For example, the following code:

```
function adder(x)
    return y->x+y
end
```

is lowered to (roughly):

```
struct ##1{T}
    x::T
end

(_::##1)(y) = _.x + y

function adder(x)
    return ##1(x)
end
```

Constructors

A constructor call is just a call to a type, to a method defined on Type{T}.

Builtins

The "builtin" functions, defined in the Core module, are:

```
<: === _abstracttype _apply_iterate _call_in_world_total _compute_sparams
_defaultctors _equiv_typedef _expr _import _primitivetype _setsuper! _structtype
_svec_len _svec_ref _typebody! _typevar _using applicable apply_type
compilerbarrier current_scope declare_const declare_global donotdelete fieldtype
finalizer get_binding_type getfield getglobal ifelse invoke invoke_in_world
invokelatest isa isdefined isdefinedglobal memorynew memoryref_isassigned
memoryrefget memoryrefmodify! memoryrefnew memoryrefoffset memoryrefreplace!
memoryrefset! memoryrefsetonce! memoryrefswap! modifyfield! modifyglobal! nfields
replacefield! replaceglobal! setfield! setfieldonce! setglobal! setglobalonce!
sizeof svec swapfield! swapglobal! throw throw_methoderror tuple typeassert
typeof
```

These are mostly singleton objects all of whose types are subtypes of Builtin, which is a subtype of Function. Their purpose is to expose entry points in the run time that use the "jlcall" calling convention:

```
jl_value_t *(jl_value_t*, jl_value_t**, uint32_t)
```

Keyword arguments

Keyword arguments work by adding methods to the kwcall function. This function is usually the "keyword argument sorter" or "keyword sorter", which then calls the inner body of the function (defined anonymously). Every definition in the kwsorter function has the same arguments as some definition in the normal method table, except with a single NamedTuple argument prepended, which gives the names and values of passed keyword arguments. The kwsorter's job is to move keyword arguments into their canonical positions based on name, plus evaluate and substitute any needed default value expressions. The result is a normal positional argument list, which is then passed to yet another compiler-generated function.

The easiest way to understand the process is to look at how a keyword argument method definition is lowered. The code:

```
function circle(center, radius; color = black, fill::Bool = true, options...)
    # draw
end
```

actually produces *three* method definitions. The first is a function that accepts all arguments (including keyword arguments) as positional arguments, and includes the code for the method body. It has an auto-generated name:

```
function #circle#1(color, fill::Bool, options, circle, center, radius)
    # draw
end
```

The second method is an ordinary definition for the original `circle` function, which handles the case where no keyword arguments are passed:

```
function circle(center, radius)
    #circle#1(black, true, pairs(NamedTuple()), circle, center, radius)
end
```

This simply dispatches to the first method, passing along default values. `pairs` is applied to the named tuple of rest arguments to provide key-value pair iteration. Note that if the method doesn't accept rest keyword arguments then this argument is absent.

Finally there is the kwsorter definition:

```
function (::Core.kwcall)(kws, circle, center, radius)
    if haskey(kws, :color)
        color = kws.color
    else
        color = black
    end
    # etc.

    # put remaining kwargs in `options`
    options = structdiff(kws, NamedTuple{(:color, :fill)})

    # if the method doesn't accept rest keywords, throw an error
    # unless `options` is empty

    #circle#1(color, fill, pairs(options), circle, center, radius)
end
```

Compiler efficiency issues

Generating a new type for every function has potentially serious consequences for compiler resource use when combined with Julia's "specialize on all arguments by default" design. Indeed, the initial implementation of this design suffered from much longer build and test times, higher memory use, and a system

image nearly 2x larger than the baseline. In a naive implementation, the problem is bad enough to make the system nearly unusable. Several significant optimizations were needed to make the design practical.

The first issue is excessive specialization of functions for different values of function-valued arguments. Many functions simply "pass through" an argument to somewhere else, e.g. to another function or to a storage location. Such functions do not need to be specialized for every closure that might be passed in. Fortunately this case is easy to distinguish by simply considering whether a function *calls* one of its arguments (i.e. the argument appears in "head position" somewhere). Performance-critical higher-order functions like `map` certainly call their argument function and so will still be specialized as expected. This optimization is implemented by recording which arguments are called during the `analyze-variables` pass in the front end. When `cache_method` sees an argument in the `Function` type hierarchy passed to a slot declared as `Any` or `Function`, it behaves as if the `@nospecialize` annotation were applied. This heuristic seems to be extremely effective in practice.

The next issue concerns the structure of method tables. Empirical studies show that the vast majority of dynamically-dispatched calls involve one or two arguments. In turn, many of these cases can be resolved by considering only the first argument. (Aside: proponents of single dispatch would not be surprised by this at all. However, this argument means "multiple dispatch is easy to optimize in practice", and that we should therefore use it, *not* "we should use single dispatch"!). So the method table and cache splits up on the structure based on a left-to-right decision tree so allow efficient nearest-neighbor searches.

The front end generates type declarations for all closures. Initially, this was implemented by generating normal type declarations. However, this produced an extremely large number of constructors, all of which were trivial (simply passing all arguments through to new). Since methods are partially ordered, inserting all of these methods is O(n²), plus there are just too many of them to keep around. This was optimized by generating `struct_type` expressions directly (bypassing default constructor generation), and using new directly to create closure instances. Not the prettiest thing ever, but you do what you gotta do.

The next problem was the `@test` macro, which generated a 0-argument closure for each test case. This is not really necessary, since each test case is simply run once in place. Therefore, `@test` was modified to expand to a try-catch block that records the test result (true, false, or exception raised) and calls the test suite handler on it.

108.9 Base.Cartesian

The (non-exported) Cartesian module provides macros that facilitate writing multidimensional algorithms. Most often you can write such algorithms with straightforward techniques; however, there are a few cases where `Base.Cartesian` is still useful or necessary.

Principles of usage

A simple example of usage is:

```
@nloops 3 i A begin
    s += @nref 3 A i
end
```

which generates the following code:

```
for i_3 = axes(A, 3)
    for i_2 = axes(A, 2)
```

```
        for i_1 = axes(A, 1)
            s += A[i_1, i_2, i_3]
        end
    end
end
```

In general, Cartesian allows you to write generic code that contains repetitive elements, like the nested loops in this example. Other applications include repeated expressions (e.g., loop unwinding) or creating function calls with variable numbers of arguments without using the "splat" construct (`i...`).

Basic syntax

The (basic) syntax of `@nloops` is as follows:

- The first argument must be an integer (*not* a variable) specifying the number of loops.
- The second argument is the symbol-prefix used for the iterator variable. Here we used `i`, and variables `i_1, i_2, i_3` were generated.
- The third argument specifies the range for each iterator variable. If you use a variable (symbol) here, it's taken as `axes(A, dim)`. More flexibly, you can use the anonymous-function expression syntax described below.
- The last argument is the body of the loop. Here, that's what appears between the `begin...end`.

There are some additional features of `@nloops` described in the reference section.

`@nref` follows a similar pattern, generating `A[i_1,i_2,i_3]` from `@nref 3 A i`. The general practice is to read from left to right, which is why `@nloops` is `@nloops 3 i A expr` (as in `for i_2 = axes(A, 2)`, where `i_2` is to the left and the range is to the right) whereas `@nref` is `@nref 3 A i` (as in `A[i_1,i_2,i_3]`, where the array comes first).

If you're developing code with Cartesian, you may find that debugging is easier when you examine the generated code, using `@macroexpand`:

```
julia> @macroexpand @nref 2 A i
:(A[i_1, i_2])
```

Supplying the number of expressions

The first argument to both of these macros is the number of expressions, which must be an integer. When you're writing a function that you intend to work in multiple dimensions, this may not be something you want to hard-code. The recommended approach is to use a `@generated function`. Here's an example:

```
@generated function mysum(A::Array{T,N}) where {T,N}
    quote
        s = zero(T)
        @nloops $N i A begin
            s += @nref $N A i
        end
        s
    end
end
```

Naturally, you can also prepare expressions or perform calculations before the `quote` block.

Anonymous-function expressions as macro arguments

Perhaps the single most powerful feature in `Cartesian` is the ability to supply anonymous-function expressions that get evaluated at parsing time. Let's consider a simple example:

```
@nexprs 2 j->(i_j = 1)
```

`@nexprs` generates `n` expressions that follow a pattern. This code would generate the following statements:

```
i_1 = 1
i_2 = 1
```

In each generated statement, an "isolated" `j` (the variable of the anonymous function) gets replaced by values in the range `1:2`. Generally speaking, Cartesian employs a LaTeX-like syntax. This allows you to do math on the index `j`. Here's an example computing the strides of an array:

```
s_1 = 1
@nexprs 3 j->(s_{j+1} = s_j * size(A, j))
```

would generate expressions

```
s_1 = 1
s_2 = s_1 * size(A, 1)
s_3 = s_2 * size(A, 2)
s_4 = s_3 * size(A, 3)
```

Anonymous-function expressions have many uses in practice.

Macro reference

`Base.Cartesian.@nloops` – Macro.

```
@nloops N itersym rangeexpr bodyexpr
@nloops N itersym rangeexpr preexpr bodyexpr
@nloops N itersym rangeexpr preexpr postexpr bodyexpr
```

Generate `N` nested loops, using `itersym` as the prefix for the iteration variables. `rangeexpr` may be an anonymous-function expression, or a simple symbol `var` in which case the range is `axes(var, d)` for dimension `d`.

Optionally, you can provide "pre" and "post" expressions. These get executed first and last, respectively, in the body of each loop. For example:

```
@nloops 2 i A d -> j_d = min(i_d, 5) begin
    s += @nref 2 A j
end
```

would generate:

```
for i_2 = axes(A, 2)
    j_2 = min(i_2, 5)
    for i_1 = axes(A, 1)
        j_1 = min(i_1, 5)
        s += A[j_1, j_2]
    end
end
```

If you want just a post-expression, supply `nothing` for the pre-expression. Using parentheses and semicolons, you can supply multi-statement expressions.

source

`Base.Cartesian.@nref` – Macro.

```
@nref N A indexexpr
```

Generate expressions like `A[i_1, i_2, ...]`. `indexexpr` can either be an iteration-symbol prefix, or an anonymous-function expression.

Examples

```
julia> @macroexpand Base.Cartesian.@nref 3 A i
:(A[i_1, i_2, i_3])
```

source

`Base.Cartesian.@nextract` – Macro.

```
@nextract N esym isym
```

Generate `N` variables `esym_1`, `esym_2`, ..., `esym_N` to extract values from `isym`. `isym` can be either a `Symbol` or anonymous-function expression.

`@nextract 2 x y` would generate

```
x_1 = y[1]
x_2 = y[2]
```

while `@nextract 3 x d->y[2d-1]` yields

```
x_1 = y[1]
x_2 = y[3]
x_3 = y[5]
```

source

`Base.Cartesian.@nexprs` – Macro.

```
@nexprs N expr
```

Generate `N` expressions. `expr` should be an anonymous-function expression.

Examples

```
julia> @macroexpand Base.Cartesian.@nexprs 4 i -> y[i] = A[i+j]
quote
    y[1] = A[1 + j]
    y[2] = A[2 + j]
    y[3] = A[3 + j]
    y[4] = A[4 + j]
end
```

source

Base.Cartesian.@ncall – Macro.

```
@ncall N f sym...
```

Generate a function call expression. `sym` represents any number of function arguments, the last of which may be an anonymous-function expression and is expanded into `N` arguments.

For example, `@ncall 3 func a` generates

```
func(a_1, a_2, a_3)
```

while `@ncall 2 func a b i->c[i]` yields

```
func(a, b, c[1], c[2])
```

source

Base.Cartesian.@ncallkw – Macro.

```
@ncallkw N f kw sym...
```

Generate a function call expression with keyword arguments `kw...`. As in the case of `@ncall`, `sym` represents any number of function arguments, the last of which may be an anonymous-function expression and is expanded into `N` arguments.

Examples

```
julia> using Base.Cartesian

julia> f(x...; a, b = 1, c = 2, d = 3) = +(x..., a, b, c, d);

julia> x_1, x_2 = (-1, -2); b = 0; kw = (c = 0, d = 0);

julia> @ncallkw 2 f (; a = 0, b, kw...) x
-3
```

source

Base.Cartesian.@ntuple – Macro.

```
@ntuple N expr
```

Generates an N-tuple. `@ntuple 2 i` would generate `(i_1, i_2)`, and `@ntuple 2 k->k+1` would generate `(2,3)`.

source

`Base.Cartesian.@nall` – Macro.

```
@nall N expr
```

Check whether all of the expressions generated by the anonymous-function expression `expr` evaluate to `true`.

`@nall 3 d->(i_d > 1)` would generate the expression `(i_1 > 1 && i_2 > 1 && i_3 > 1)`. This can be convenient for bounds-checking.

source

`Base.Cartesian.@nany` – Macro.

```
@nany N expr
```

Check whether any of the expressions generated by the anonymous-function expression `expr` evaluate to `true`.

`@nany 3 d->(i_d > 1)` would generate the expression `(i_1 > 1 || i_2 > 1 || i_3 > 1)`.

source

`Base.Cartesian.@nif` – Macro.

```
@nif N conditionexpr expr
@nif N conditionexpr expr elseexpr
```

Generates a sequence of `if ... elseif ... else ... end` statements. For example:

```
@nif 3 d->(i_d >= size(A,d)) d->(error("Dimension ", d, " too big")) d->println("All OK")
```

would generate:

```
if i_1 > size(A, 1)
    error("Dimension ", 1, " too big")
elseif i_2 > size(A, 2)
    error("Dimension ", 2, " too big")
else
    println("All OK")
end
```

source

108.10 Talking to the compiler (the `:meta` mechanism)

In some circumstances, one might wish to provide hints or instructions that a given block of code has special properties: you might always want to inline it, or you might want to turn on special compiler optimization passes. Starting with version 0.4, Julia has a convention that these instructions can be placed inside a `:meta` expression, which is typically (but not necessarily) the first expression in the body of a function.

`:meta` expressions are created with macros. As an example, consider the implementation of the `@inline` macro:

```
macro inline(ex)
    esc(isa(ex, Expr) ? pushmeta!(ex, :inline) : ex)
end
```

Here, ex is expected to be an expression defining a function. A statement like this:

```
@inline function myfunction(x)
    x*(x+3)
end
```

gets turned into an expression like this:

```
quote
    function myfunction(x)
        Expr(:meta, :inline)
        x*(x+3)
    end
end
```

`Base.pushmeta!(ex, tag::Union{Symbol,Expr})` appends `:tag` to the end of the `:meta` expression, creating a new `:meta` expression if necessary.

To use the metadata, you have to parse these `:meta` expressions. If your implementation can be performed within Julia, `Base.popmeta!` is very handy: `Base.popmeta!(body, :symbol)` will scan a function *body* expression (one without the function signature) for the first `:meta` expression containing `:symbol`, extract any arguments, and return a tuple `(found::Bool, args::Array{Any})`. If the metadata did not have any arguments, or `:symbol` was not found, the `args` array will be empty.

Not yet provided is a convenient infrastructure for parsing `:meta` expressions from C++.

108.11 SubArrays

Julia's `SubArray` type is a container encoding a "view" of a parent `AbstractArray`. This page documents some of the design principles and implementation of `SubArrays`.

One of the major design goals is to ensure high performance for views of both `IndexLinear` and `IndexCartesian` arrays. Furthermore, views of `IndexLinear` arrays should themselves be `IndexLinear` to the extent that it is possible.

Index replacement

Consider making 2d slices of a 3d array:

```
julia> A = rand(2,3,4);

julia> S1 = view(A, :, 1, 2:3)
2×2 view(::Array{Float64, 3}, :, 1, 2:3) with eltype Float64:
```

```
 0.839622  0.711389
 0.967143  0.103929

julia> S2 = view(A, 1, :, 2:3)
3×2 view(::Array{Float64, 3}, 1, :, 2:3) with eltype Float64:
 0.839622  0.711389
 0.789764  0.806704
 0.566704  0.962715
```

`view` drops "singleton" dimensions (ones that are specified by an `Int`), so both `S1` and `S2` are two-dimensional `SubArrays`. Consequently, the natural way to index these is with `S1[i,j]`. To extract the value from the parent array `A`, the natural approach is to replace `S1[i,j]` with `A[i,1,(2:3)[j]]` and `S2[i,j]` with `A[1,i,(2:3)[j]]`.

The key feature of the design of SubArrays is that this index replacement can be performed without any runtime overhead.

SubArray design

Type parameters and fields

The strategy adopted is first and foremost expressed in the definition of the type:

```
struct SubArray{T,N,P,I,L} <: AbstractArray{T,N}
    parent::P
    indices::I
    offset1::Int       # for linear indexing and pointer, only valid when L==true
    stride1::Int       # used only for linear indexing
    ...
end
```

`SubArray` has 5 type parameters. The first two are the standard element type and dimensionality. The next is the type of the parent `AbstractArray`. The most heavily-used is the fourth parameter, a `Tuple` of the types of the indices for each dimension. The final one, `L`, is only provided as a convenience for dispatch; it's a boolean that represents whether the index types support fast linear indexing. More on that later.

If in our example above `A` is a `Array{Float64, 3}`, our `S1` case above would be a `SubArray{Float64,2,Array{Float64,3},Tuple`
Note in particular the tuple parameter, which stores the types of the indices used to create `S1`. Likewise,

```
julia> S1.indices
(Base.Slice(Base.OneTo(2)), 1, 2:3)
```

Storing these values allows index replacement, and having the types encoded as parameters allows one to dispatch to efficient algorithms.

Index translation

Performing index translation requires that you do different things for different concrete `SubArray` types. For example, for `S1`, one needs to apply the `i,j` indices to the first and third dimensions of the parent array, whereas for `S2` one needs to apply them to the second and third. The simplest approach to indexing would be to do the type-analysis at runtime:

```
parentindices = Vector{Any}()
for thisindex in S.indices
    ...
    if isa(thisindex, Int)
        # Don't consume one of the input indices
        push!(parentindices, thisindex)
    elseif isa(thisindex, AbstractVector)
        # Consume an input index
        push!(parentindices, thisindex[inputindex[j]])
        j += 1
    elseif isa(thisindex, AbstractMatrix)
        # Consume two input indices
        push!(parentindices, thisindex[inputindex[j], inputindex[j+1]])
        j += 2
    elseif ...
end
S.parent[parentindices...]
```

Unfortunately, this would be disastrous in terms of performance: each element access would allocate memory, and involves the running of a lot of poorly-typed code.

The better approach is to dispatch to specific methods to handle each type of stored index. That's what `reindex` does: it dispatches on the type of the first stored index and consumes the appropriate number of input indices, and then it recurses on the remaining indices. In the case of `S1`, this expands to

```
Base.reindex(S1, S1.indices, (i, j)) == (i, S1.indices[2], S1.indices[3][j])
```

for any pair of indices `(i,j)` (except `CartesianIndex`s and arrays thereof, see below).

This is the core of a `SubArray`; indexing methods depend upon `reindex` to do this index translation. Sometimes, though, we can avoid the indirection and make it even faster.

Linear indexing

Linear indexing can be implemented efficiently when the entire array has a single stride that separates successive elements, starting from some offset. This means that we can pre-compute these values and represent linear indexing simply as an addition and multiplication, avoiding the indirection of `reindex` and (more importantly) the slow computation of the cartesian coordinates entirely.

For `SubArray` types, the availability of efficient linear indexing is based purely on the types of the indices, and does not depend on values like the size of the parent array. You can ask whether a given set of indices supports fast linear indexing with the internal `Base.viewindexing` function:

```
julia> Base.viewindexing(S1.indices)
IndexCartesian()

julia> Base.viewindexing(S2.indices)
IndexLinear()
```

This is computed during construction of the `SubArray` and stored in the `L` type parameter as a boolean that encodes fast linear indexing support. While not strictly necessary, it means that we can define dispatch directly on `SubArray{T,N,A,I,true}` without any intermediaries.

Since this computation doesn't depend on runtime values, it can miss some cases in which the stride happens to be uniform:

```
julia> A = reshape(1:4*2, 4, 2)
4×2 reshape(::UnitRange{Int64}, 4, 2) with eltype Int64:
 1  5
 2  6
 3  7
 4  8

julia> diff(A[2:2:4,:][:])
3-element Vector{Int64}:
 2
 2
 2
```

A view constructed as `view(A, 2:2:4, :)` happens to have uniform stride, and therefore linear indexing indeed could be performed efficiently. However, success in this case depends on the size of the array: if the first dimension instead were odd,

```
julia> A = reshape(1:5*2, 5, 2)
5×2 reshape(::UnitRange{Int64}, 5, 2) with eltype Int64:
 1   6
 2   7
 3   8
 4   9
 5  10

julia> diff(A[2:2:4,:][:])
3-element Vector{Int64}:
 2
 3
 2
```

then `A[2:2:4,:]` does not have uniform stride, so we cannot guarantee efficient linear indexing. Since we have to base this decision based purely on types encoded in the parameters of the `SubArray`, `S = view(A, 2:2:4, :)` cannot implement efficient linear indexing.

A few details

- Note that the `Base.reindex` function is agnostic to the types of the input indices; it simply determines how and where the stored indices should be reindexed. It not only supports integer indices, but it supports non-scalar indexing, too. This means that views of views don't need two levels of indirection; they can simply re-compute the indices into the original parent array!
- Hopefully by now it's fairly clear that supporting slices means that the dimensionality, given by the parameter `N`, is not necessarily equal to the dimensionality of the parent array or the length of the `indices` tuple. Neither do user-supplied indices necessarily line up with entries in the `indices` tuple (e.g., the second user-supplied index might correspond to the third dimension of the parent array, and the third element in the `indices` tuple).

What might be less obvious is that the dimensionality of the stored parent array must be equal to the number of effective indices in the `indices` tuple. Some examples:

```
A = reshape(1:35, 5, 7) # A 2d parent Array
S = view(A, 2:7)         # A 1d view created by linear indexing
S = view(A, :, :, 1:1)   # Appending extra indices is supported
```

Naively, you'd think you could just set `S.parent = A` and `S.indices = (:,:,1:1)`, but supporting this dramatically complicates the reindexing process, especially for views of views. Not only do you need to dispatch on the types of the stored indices, but you need to examine whether a given index is the final one and "merge" any remaining stored indices together. This is not an easy task, and even worse: it's slow since it implicitly depends upon linear indexing.

Fortunately, this is precisely the computation that `ReshapedArray` performs, and it does so linearly if possible. Consequently, `view` ensures that the parent array is the appropriate dimensionality for the given indices by reshaping it if needed. The inner `SubArray` constructor ensures that this invariant is satisfied.

- `CartesianIndex` and arrays thereof throw a nasty wrench into the `reindex` scheme. Recall that `reindex` simply dispatches on the type of the stored indices in order to determine how many passed indices should be used and where they should go. But with `CartesianIndex`, there's no longer a one-to-one correspondence between the number of passed arguments and the number of dimensions that they index into. If we return to the above example of `Base.reindex(S1, S1.indices, (i, j))`, you can see that the expansion is incorrect for `i, j = CartesianIndex(), CartesianIndex(2,1)`. It should *skip* the `CartesianIndex()` entirely and return:

  ```
  (CartesianIndex(2,1)[1], S1.indices[2], S1.indices[3][CartesianIndex(2,1)[2]])
  ```

  Instead, though, we get:

  ```
  (CartesianIndex(), S1.indices[2], S1.indices[3][CartesianIndex(2,1)])
  ```

  Doing this correctly would require *combined* dispatch on both the stored and passed indices across all combinations of dimensionalities in an intractable manner. As such, `reindex` must never be called with `CartesianIndex` indices. Fortunately, the scalar case is easily handled by first flattening the `CartesianIndex` arguments to plain integers. Arrays of `CartesianIndex`, however, cannot be split apart into orthogonal pieces so easily. Before attempting to use `reindex`, `view` must ensure that there are no arrays of `CartesianIndex` in the argument list. If there are, it can simply "punt" by avoiding the `reindex` calculation entirely, constructing a nested `SubArray` with two levels of indirection instead.

108.12 isbits Union Optimizations

In Julia, the `Array` type holds both "bits" values as well as heap-allocated "boxed" values. The distinction is whether the value itself is stored inline (in the direct allocated memory of the array), or if the memory of the array is simply a collection of pointers to objects allocated elsewhere. In terms of performance, accessing values inline is clearly an advantage over having to follow a pointer to the actual value. The definition of "isbits" generally means any Julia type with a fixed, determinate size, meaning no "pointer" fields, see `?isbitstype`.

Julia also supports Union types, quite literally the union of a set of types. Custom Union type definitions can be extremely handy for applications wishing to "cut across" the nominal type system (i.e. explicit subtype

relationships) and define methods or functionality on these, otherwise unrelated, set of types. A compiler challenge, however, is in determining how to treat these Union types. The naive approach (and indeed, what Julia itself did pre-0.7), is to simply make a "box" and then a pointer in the box to the actual value, similar to the previously mentioned "boxed" values. This is unfortunate, however, because of the number of small, primitive "bits" types (think `UInt8`, `Int32`, `Float64`, etc.) that would easily fit themselves inline in this "box" without needing any indirection for value access. There are two main ways Julia can take advantage of this optimization as of 0.7: isbits Union fields in types, and isbits Union Arrays.

isbits Union Structs

Julia now includes an optimization wherein "isbits Union" fields in types (`mutable struct`, `struct`, etc.) will be stored inline. This is accomplished by determining the "inline size" of the Union type (e.g. `Union{UInt8, Int16}` will have a size of two bytes, which represents the size needed of the largest Union type `Int16`), and in addition, allocating an extra "type tag byte" (`UInt8`), whose value signals the type of the actual value stored inline of the "Union bytes". The type tag byte value is the index of the actual value's type in the Union type's order of types. For example, a type tag value of `0x02` for a field with type `Union{Nothing, UInt8, Int16}` would indicate that an `Int16` value is stored in the 16 bits of the field in the structure's memory; a `0x01` value would indicate that a `UInt8` value was stored in the first 8 bits of the 16 bits of the field's memory. Lastly, a value of `0x00` signals that the `nothing` value will be returned for this field, even though, as a singleton type with a single type instance, it technically has a size of 0. The type tag byte for a type's Union field is stored directly after the field's computed Union memory.

isbits Union Memory

Julia can now also store "isbits Union" values inline in a Memory, as opposed to requiring an indirection box. The optimization is accomplished by storing an extra "type tag memory" of bytes, one byte per element, alongside the bytes of the actual data. This type tag memory serves the same function as the type field case: its value signals the type of the actual stored Union value. The "type tag memory" directly follows the regular data space. So the formula to access an isbits Union Array's type tag bytes is `a->data + a->length * a->elsize`.

108.13 System Image Building

Building the Julia system image

Julia ships with a preparsed system image containing the contents of the `Base` module, named `sys.ji`. This file is also precompiled into a shared library called `sys.{so,dll,dylib}` on as many platforms as possible, so as to give vastly improved startup times. On systems that do not ship with a precompiled system image file, one can be generated from the source files shipped in Julia's `DATAROOTDIR/julia/base` folder.

Julia will by default generate its system image on half of the available system threads. This may be controlled by the `JULIA_IMAGE_THREADS` environment variable.

This operation is useful for multiple reasons. A user may:

- Build a precompiled shared library system image on a platform that did not ship with one, thereby improving startup times.
- Modify `Base`, rebuild the system image and use the new `Base` next time Julia is started.

- Include a `userimg.jl` file that includes packages into the system image, thereby creating a system image that has packages embedded into the startup environment.

The `PackageCompiler.jl package` contains convenient wrapper functions to automate this process.

System image optimized for multiple microarchitectures

The system image can be compiled simultaneously for multiple CPU microarchitectures under the same instruction set architecture (ISA). Multiple versions of the same function may be created with minimum dispatch point inserted into shared functions in order to take advantage of different ISA extensions or other microarchitecture features. The version that offers the best performance will be selected automatically at runtime based on available CPU features.

Specifying multiple system image targets

A multi-microarchitecture system image can be enabled by passing multiple targets during system image compilation. This can be done either with the `JULIA_CPU_TARGET` make option or with the `-C` command line option when running the compilation command manually. Multiple targets are separated by `;` in the option string. The syntax for each target is a CPU name followed by multiple features separated by `,`. All features supported by LLVM are supported and a feature can be disabled with a `-` prefix. (`+` prefix is also allowed and ignored to be consistent with LLVM syntax). Additionally, a few special features are supported to control the function cloning behavior.

Note

It is good practice to specify either `clone_all` or `base(<n>)` for every target apart from the first one. This makes it explicit which targets have all functions cloned, and which targets are based on other targets. If this is not done, the default behavior is to not clone every function, and to use the first target's function definition as the fallback when not cloning a function.

1. `clone_all`

   By default, only functions that are the most likely to benefit from the microarchitecture features will be cloned. When `clone_all` is specified for a target, however, **all** functions in the system image will be cloned for the target. The negative form `-clone_all` can be used to prevent the built-in heuristic from cloning all functions.

2. `base(<n>)`

   Where `<n>` is a placeholder for a non-negative number (e.g. `base(0)`, `base(1)`). By default, a partially cloned (i.e. not `clone_all`) target will use functions from the default target (first one specified) if a function is not cloned. This behavior can be changed by specifying a different base with the `base(<n>)` option. The nth target (0-based) will be used as the base target instead of the default (`0`th) one. The base target has to be either `0` or another `clone_all` target. Specifying a non-`clone_all` target as the base target will cause an error.

3. `opt_size`

   This causes the function for the target to be optimized for size when there isn't a significant runtime performance impact. This corresponds to `-Os` GCC and Clang option.

4. `min_size`

   This causes the function for the target to be optimized for size that might have a significant runtime performance impact. This corresponds to `-Oz` Clang option.

As an example, at the time of this writing, the following string is used in the creation of the official x86_64 Julia binaries downloadable from julialang.org:

```
generic;sandybridge,-xsaveopt,clone_all;haswell,-rdrnd,base(1)
```

This creates a system image with three separate targets; one for a generic x86_64 processor, one with a `sandybridge` ISA (explicitly excluding `xsaveopt`) that explicitly clones all functions, and one targeting the `haswell` ISA, based off of the `sandybridge` sysimg version, and also excluding `rdrnd`. When a Julia implementation loads the generated sysimg, it will check the host processor for matching CPU capability flags, enabling the highest ISA level possible. Note that the base level (`generic`) requires the `cx16` instruction, which is disabled in some virtualization software and must be enabled for the `generic` target to be loaded. Alternatively, a sysimg could be generated with the target `generic,-cx16` for greater compatibility, however note that this may cause performance and stability problems in some code.

Implementation overview

This is a brief overview of different part involved in the implementation. See code comments for each components for more implementation details.

1. System image compilation

   The parsing and cloning decision are done in `src/processor*`. We currently support cloning of function based on the present of loops, simd instructions, or other math operations (e.g. fast-math, fma, muladd). This information is passed on to `src/llvm-multiversioning.cpp` which does the actual cloning. In addition to doing the cloning and insert dispatch slots (see comments in `MultiVersioning::runOnModule` for how this is done), the pass also generates metadata so that the runtime can load and initialize the system image correctly. A detailed description of the metadata is available in `src/processor.h`.

2. System image loading

   The loading and initialization of the system image is done in `src/processor*` by parsing the metadata saved during system image generation. Host feature detection and selection decision are done in `src/processor_*.cpp` depending on the ISA. The target selection will prefer exact CPU name match, larger vector register size, and larger number of features. An overview of this process is in `src/processor.cpp`.

Trimming

System images are typically quite large, since Base includes a lot of functionality, and by default system images also include several packages such as LinearAlgebra for convenience and backwards compatibility. Most programs will use only a fraction of the functions in these packages. Therefore it makes sense to build binaries that exclude unused functions to save space, referred to as "trimming".

While the basic idea of trimming is sound, Julia has dynamic and reflective features that make it difficult (or impossible) to know in general which functions are unused. As an extreme example, consider code like

```
getglobal(Base, Symbol(readchomp(stdin)))(1)
```

This code reads a function name from `stdin` and calls the named function from Base on the value 1. In this case it is impossible to predict which function will be called, so no functions can reliably be considered "unused". With some noteworthy exceptions (Julia's own REPL being one of them), most real-world programs do not do things like this.

Less extreme cases occur, for example, when there are type instabilities that make it impossible for the compiler to predict which method will be called. However, if code is well-typed and does not use reflection, a complete and (hopefully) relatively small set of needed methods can be determined, and the rest can be removed. The `--trim` command-line option requests this kind of compilation.

When `--trim` is specified in a command used to build a system image, the compiler begins tracing calls starting at methods marked using `Base.Experimental.entrypoint`. If a call is too dynamic to reasonably narrow down the possible call targets, an error is given at compile time showing the location of the call. For testing purposes, it is possible to skip these errors by specifying `--trim=unsafe` or `--trim=unsafe-warn`. Then you will get a system image built, but it may crash at run time if needed code is not present.

It typically makes sense to specify `--strip-ir` along with `--trim`, since trimmed binaries are fully compiled and therefore don't need Julia IR. At some point we may make `--trim` imply `--strip-ir`, but for now we have kept them orthogonal.

To get the smallest possible binary, it will also help to specify `--strip-metadata` and run the Unix `strip` utility. However, those steps remove Julia-specific and native (DWARF format) debug info, respectively, and so will make debugging more difficult.

Rather than invoking these options by hand, most users should build trimmed binaries with JuliaC.jl, which supplies the `juliac` driver and handles compiling, linking, and bundling the result.

Common problems

- The Base global variables `stdin`, `stdout`, and `stderr` are non-constant and so their types are not known. All printing should use a specific IO object with a known type. The easiest substitution is to use `print(Core.stdout, x)` instead of `print(x)` or `print(stdout, x)`.
- Use tools like JET.jl, Cthulhu.jl, and/or SnoopCompile to identify failures of type-inference, and follow our Performance Tips to fix them.

Compatibility concerns

We have identified many small changes to Base that significantly increase the set of programs that can be reliably trimmed. Unfortunately some of those changes would be considered breaking, and so are only applied when trimming is requested (this is done by an external build script, currently maintained inside the test suite as `test/trimming/juliac-buildscript.jl`). Therefore in many cases trimming will require you to opt in to new variants of Base and some standard libraries.

If you want to use trimming, it is important to set up continuous integration testing that performs a trimmed build and fully tests the resulting program. Fortunately, if your program successfully compiles with `--trim` then it is very likely to work the same as it did before. However, CI is needed to ensure that your program continues to build with trimming as you develop it.

Package authors may wish to test that their package is "trimming safe", however this is impossible in general. Trimming is only expected to work given concrete entry points such as `main()` and library entry

points meant to be called from outside Julia. For generic packages, existing tests for type stability like `@inferred` and `JET.@report_call` are about as close as you can get to checking trim compatibility.

Trimming also introduces new compatibility issues between minor versions of Julia. At this time, we are not able to guarantee that a program that can be trimmed in one version of Julia can also be trimmed in all future versions of Julia. However, breakage of that kind is expected to be rare. We also plan to try to *increase* the set of programs that can be trimmed over time.

108.14 Package Images

Julia package images provide object (native code) caches for Julia packages. They are similar to Julia's system image and support many of the same features. In fact the underlying serialization format is the same, and the system image is the base image that the package images are build against.

High-level overview

Package images are shared libraries that contain both code and data. Like `.ji` cache files, they are generated per package. The data section contains both global data (global variables in the package) as well as the necessary metadata about what methods and types are defined by the package. The code section contains native objects that cache the final output of Julia's LLVM-based compiler.

The command line option `--pkgimages=no` can be used to turn off object caching for this session. Note that this means that cache files have to likely be regenerated. See `JULIA_MAX_NUM_PRECOMPILE_FILES` for the upper limit of variants Julia caches per default.

Note

While the package images present themselves as native shared libraries, they are only an approximation thereof. You will not be able to link against them from a native program and they must be loaded from Julia.

Linking

Since the package images contain native code, we must run a linker over them before we can use them. You can set the environment variable `JULIA_VERBOSE_LINKING` to `true` to make the package image linking process verbose.

Furthermore, we cannot assume that the user has a working system linker installed. Therefore, Julia ships with LLD, the LLVM linker, to provide a working out of the box experience. In `base/linking.jl`, we implement a limited interface to be able to link package images on all supported platforms.

Quirks

Despite LLD being a multi-platform linker, it does not provide a consistent interface across platforms. Furthermore, it is meant to be used from `clang` or another compiler driver, we therefore reimplement some of the logic from `llvm-project/clang/lib/Driver/ToolChains`. Thankfully one can use `lld -flavor` to set lld to the right platform

Windows

To avoid having to deal with `link.exe` we use `-flavor gnu`, effectively turning `lld` into a cross-linker from a mingw32 environment. Windows DLLs are required to contain a `_DllMainCRTStartup` function and to minimize our dependence on mingw32 libraries, we inject a stub definition ourselves.

MacOS

Dynamic libraries on macOS need to link against `-lSystem`. On recent macOS versions, `-lSystem` is only available for linking when Xcode is available. To that effect we link with `-undefined dynamic_lookup`.

Package images optimized for multiple microarchitectures

Similar to multi-versioning for system images, package images support multi-versioning. This allows creating package caches that can run efficiently on different CPU architectures within the same environment.

See the `JULIA_CPU_TARGET` environment variable for more information on how to set the CPU target for package images.

Flags that impact package image creation and selection

These are the Julia command line flags that impact cache selection. Package images that were created with different flags will be rejected.

- `-g`, `--debug-info`: Exact match required since it changes code generation.
- `--check-bounds`: Exact match required since it changes code generation.
- `--inline`: Exact match required since it changes code generation.
- `--pkgimages`: To allow running without object caching enabled.
- `-O`, `--optimize`: Reject package images generated for a lower optimization level, but allow for higher optimization levels to be loaded.

108.15 Custom LLVM Passes

Julia has a number of custom LLVM passes. Broadly, they can be classified into passes that are required to be run to maintain Julia semantics, and passes that take advantage of Julia semantics to optimize LLVM IR.

Semantic Passes

These passes are used to transform LLVM IR into code that is legal to be run on a CPU. Their main purpose is to enable simpler IR to be emitted by codegen, which then enables other LLVM passes to optimize common patterns.

CPUFeatures

- Filename: `llvm-cpufeatures.cpp`
- Class Name: `CPUFeaturesPass`
- Opt Name: `module(CPUFeatures)`

This pass lowers the `julia.cpu.have_fma.(f32|f64)` intrinsic to either true or false, depending on the target architecture and target features present on the function. This intrinsic is often used to determine if using algorithms dependent on fast fused multiply-add operations is better than using standard algorithms not dependent on such instructions.

DemoteFloat16

- Filename: `llvm-demote-float16.cpp`
- ClassName: `DemoteFloat16Pass`
- Opt Name `function(DemoteFloat16)`

This pass replaces float16 operations with float32 operations on architectures that do not natively support float16 operations. This is done by inserting `fpext` and `fptrunc` instructions around any float16 operation. On architectures that do support native float16 operations, this pass is a no-op.

LateGCLowering

- Filename: `llvm-late-gc-lowering.cpp`
- Class Name: `LateLowerGCPass`
- Opt Name: `function(LateLowerGCFrame)`

This pass performs most of the GC rooting work required to track pointers between GC safepoints. It also lowers several intrinsics to their corresponding instruction translation, and is permitted to violate the non-integral invariants previously established (`pointer_from_objref` is lowered to a `ptrtoint` instruction here). This pass typically occupies the most time out of all the custom Julia passes, due to its dataflow algorithm to minimize the number of objects live at any safepoint.

FinalGCLowering

- Filename: `llvm-final-gc-lowering.cpp`
- Class Name: `FinalLowerGCPass`
- Opt Name: `module(FinalLowerGC)`

This pass lowers a few last intrinsics to their final form targeting functions in the `libjulia` library. Separating this from `LateGCLowering` enables other backends (GPU compilation) to supply their own custom lowerings for these intrinsics, enabling the Julia pipeline to be used on those backends as well.

RemoveNI

- Filename: `llvm-remove-ni.cpp`
- Class Name: `RemoveNIPass`
- Opt Name: `module(RemoveNI)`

This pass removes the non-integral address spaces from the module's datalayout string. This enables the backend to lower Julia's custom address spaces directly to machine code, without a costly rewrite of every pointer operation to address space 0.

SIMDLoop

- Filename: `llvm-simdloop.cpp`
- Class Name: `LowerSIMDLoopPass`
- Opt Name: `loop(LowerSIMDLoop)`

This pass acts as the main driver of the `@simd` annotation. Codegen inserts a `!llvm.loopid` marker at the back branch of a loop, which this pass uses to identify loops that were originally marked with `@simd`. Then, this pass looks for a chain of floating point operations that form a reduce and adds the `contract` and `reassoc` fast math flags to allow reassociation (and thus vectorization). This pass does not preserve either loop information nor inference correctness, so it may violate Julia semantics in surprising ways. If the loop was annotated with `ivdep` as well, then the pass marks the loop as having no loop-carried dependencies (the resulting behavior is undefined if the user annotation was incorrect or gets applied to the wrong loop).

LowerPTLS

- Filename: `llvm-ptls.cpp`
- Class Name: `LowerPTLSPass`
- Opt Name: `module(LowerPTLSPass)`

This pass lowers thread-local Julia intrinsics to assembly instructions. Julia relies on thread-local storage for garbage collection and multithreading task scheduling. When compiling code for system images and package images, this pass replaces calls to intrinsics with loads from global variables that are initialized at load time.

If codegen produces a function with a `swiftself` argument and calling convention, this pass assumes the `swiftself` argument is the pgcstack and will replace the intrinsics with that argument. Doing so provides speedups on architectures that have slow thread local storage accesses.

RemoveAddrspaces

- Filename: `llvm-remove-addrspaces.cpp`
- Class Name: `RemoveAddrspacesPass`
- Opt Name: `module(RemoveAddrspaces)`

This pass renames pointers in one address space to another address space. This is used to remove Julia-specific address spaces from LLVM IR.

RemoveJuliaAddrspaces

- Filename: `llvm-remove-addrspaces.cpp`
- Class Name: `RemoveJuliaAddrspacesPass`
- Opt Name: `module(RemoveJuliaAddrspaces)`

This pass removes Julia-specific address spaces from LLVM IR. It is mostly used for displaying LLVM IR in a less cluttered format. Internally, it is implemented off the RemoveAddrspaces pass.

Multiversioning

- Filename: `llvm-multiversioning.cpp`
- Class Name: `MultiVersioningPass`
- Opt Name: `module(JuliaMultiVersioning)`

This pass performs modifications to a module to create functions that are optimized for running on different architectures (see sysimg.md and pkgimg.md for more details). Implementation-wise, it clones functions and applies different target-specific attributes to them to allow the optimizer to use advanced features such as vectorization and instruction scheduling for that platform. It also creates some infrastructure to enable the Julia image loader to select the appropriate version of the function to call based on the architecture the loader is running on. The target-specific attributes are controlled by the `julia.mv.specs` module flag, which during compilation is derived from the `JULIA_CPU_TARGET` environment variable. The pass must also be enabled by providing a `julia.mv.enable` module flag with a value of 1.

Warning

Use of `llvmcall` with multiversioning is dangerous. `llvmcall` enables access to features not typically exposed by the Julia APIs, and are therefore usually not available on all architectures. If multiversioning is enabled and code generation is requested for a target architecture that does not support the feature required by an `llvmcall` expression, LLVM will probably error out, likely with an abort and the message `LLVM ERROR: Do not know how to split the result of this operator!`.

GCInvariantVerifier

- Filename: `llvm-gc-invariant-verifier.cpp`
- Class Name: `GCInvariantVerifierPass`
- Opt Name: `module(GCInvariantVerifier)`

This pass is used to verify Julia's invariants about LLVM IR. This includes things such as the nonexistence of `ptrtoint` in Julia's non-integral address spaces [1] and the existence of only blessed `addrspacecast` instructions (Tracked -> Derived, 0 -> Tracked, etc). It performs no transformations on IR.

Optimization Passes

These passes are used to perform transformations on LLVM IR that LLVM will not perform itself, e.g. fast math flag propagation, escape analysis, and optimizations on Julia-specific internal functions. They use knowledge about Julia's semantics to perform these optimizations.

AllocOpt

- Filename: `llvm-alloc-opt.cpp`
- Class Name: `AllocOptPass`

[1] https://llvm.org/devmtg/2015-02/slides/chisnall-pointers-not-int.pdf

- Opt Name: `function(AllocOpt)`

Julia does not have the concept of a program stack as a place to allocate mutable objects. However, allocating objects on the stack reduces GC pressure and is critical for GPU compilation. Thus, `AllocOpt` performs heap to stack conversion of objects that it can prove do not escape the current function. It also performs a number of other optimizations on allocations, such as removing allocations that are never used, optimizing typeof calls to freshly allocated objects, and removing stores to allocations that are immediately overwritten. The escape analysis implementation is located in `llvm-alloc-helpers.cpp`. Currently, this pass does not use information from `EscapeAnalysis.jl`, though that may change in the future.

PropagateJuliaAddrspaces

- Filename: `llvm-propagate-addrspaces.cpp`
- Class Name: `PropagateJuliaAddrspacesPass`
- Opt Name: `function(PropagateJuliaAddrspaces)`

This pass is used to propagate Julia-specific address spaces through operations on pointers. LLVM is not allowed to introduce or remove addrspacecast instructions by optimizations, so this pass acts to eliminate redundant addrspace casts by replacing operations with their equivalent in a Julia address space. For more information on Julia's address spaces, see (TODO link to llvm.md).

JuliaLICM

- Filename: `llvm-julia-licm.cpp`
- Class Name: `JuliaLICMPass`
- Opt Name: `loop(JuliaLICM)`

This pass is used to hoist Julia-specific intrinsics out of loops. Specifically, it performs the following transformations:

1. Hoist `gc_preserve_begin` and sink `gc_preserve_end` out of loops when the preserved objects are loop-invariant.
    1. Since objects preserved within a loop are likely preserved for the duration of the loop, this transformation can reduce the number of `gc_preserve_begin`/`gc_preserve_end` pairs in the IR. This makes it easier for the `LateLowerGCPass` to identify where particular objects are preserved.
2. Hoist write barriers with invariant objects
    1. Here we assume that there are only two generations that an object can be a part of. Given that, a write barrier needs to only execute once for any pair of the same object. Thus, we can hoist write barriers out of loops when the object being written to is loop-invariant.
3. Hoist allocations out of loops when they do not escape the loop
    1. We use a very conservative definition of escape here, the same as the one used in `AllocOptPass`. This transformation can reduce the number of allocations in the IR, even when an allocation escapes the function altogether.

Note

This pass is required to preserve LLVM's MemorySSA (Short Video, Longer Video) and ScalarEvolution (Newer Slides Older Slides) analyses.

108.16 Working with LLVM

This is not a replacement for the LLVM documentation, but a collection of tips for working on LLVM for Julia.

Overview of Julia to LLVM Interface

Julia dynamically links against LLVM by default. Build with `USE_LLVM_SHLIB=0` to link statically.

The code for lowering Julia AST to LLVM IR or interpreting it directly is in directory `src/`.

File	Description
`aotcompile.cpp`	Compiler C-interface entry and object file emission
`builtins.c`	Builtin functions
`ccall.cpp`	Lowering `ccall`
`cgutils.cpp`	Lowering utilities, notably for array and tuple accesses
`codegen.cpp`	Top-level of code generation, pass list, lowering builtins
`debuginfo.cpp`	Tracks debug information for JIT code
`disasm.cpp`	Handles native object file and JIT code disassembly
`gf.c`	Generic functions
`intrinsics.cpp`	Lowering intrinsics
`jitlayers.cpp`	JIT-specific code, ORC compilation layers/utilities
`llvm-alloc-helpers.cpp`	Julia-specific escape analysis
`llvm-alloc-opt.cpp`	Custom LLVM pass to demote heap allocations to the stack
`llvm-cpufeatures.cpp`	Custom LLVM pass to lower CPU-based functions (e.g. haveFMA)
`llvm-demote-float16.cpp`	Custom LLVM pass to lower 16b float ops to 32b float ops
`llvm-final-gc-lowering.cpp`	Custom LLVM pass to lower GC calls to their final form
`llvm-gc-invariant-verifier.cpp`	Custom LLVM pass to verify Julia GC invariants
`llvm-julia-licm.cpp`	Custom LLVM pass to hoist/sink Julia-specific intrinsics
`llvm-late-gc-lowering.cpp`	Custom LLVM pass to root GC-tracked values
`llvm-multiversioning.cpp`	Custom LLVM pass to generate sysimg code on multiple architectures
`llvm-propagate-addrspaces.cpp`	Custom LLVM pass to canonicalize addrspaces
`llvm-ptls.cpp`	Custom LLVM pass to lower TLS operations
`llvm-remove-addrspaces.cpp`	Custom LLVM pass to remove Julia addrspaces
`llvm-remove-ni.cpp`	Custom LLVM pass to remove Julia non-integral addrspaces
`llvm-simdloop.cpp`	Custom LLVM pass for `@simd`
`pipeline.cpp`	New pass manager pipeline, pass pipeline parsing
`sys.c`	I/O and operating system utility functions

Some of the `.cpp` files form a group that compile to a single object.

The difference between an intrinsic and a builtin is that a builtin is a first class function that can be used like any other Julia function. An intrinsic can operate only on unboxed data, and therefore its arguments must be statically typed.

Alias Analysis

Julia currently uses LLVM's Type Based Alias Analysis. To find the comments that document the inclusion relationships, look for `static MDNode*` in `src/codegen.cpp`.

The `-O` option enables LLVM's Basic Alias Analysis.

Building Julia with a different version of LLVM

The default version of LLVM is specified in `deps/llvm.version`. You can override it by creating a file called `Make.user` in the top-level directory and adding a line to it such as:

```
LLVM_VER = 13.0.0
```

Besides the LLVM release numerals, you can also use `DEPS_GIT = llvm` in combination with `USE_BINARYBUILDER_LLVM = 0` to build against the latest development version of LLVM.

You can also specify to build a debug version of LLVM, by setting either `LLVM_DEBUG = 1` or `LLVM_DEBUG = Release` in your `Make.user` file. The former will be a fully unoptimized build of LLVM and the latter will produce an optimized build of LLVM. Depending on your needs the latter will suffice and it quite a bit faster. If you use `LLVM_DEBUG = Release` you will also want to set `LLVM_ASSERTIONS = 1` to enable diagnostics for different passes. Only `LLVM_DEBUG = 1` implies that option by default.

Passing options to LLVM

You can pass options to LLVM via the environment variable `JULIA_LLVM_ARGS`. Here are example settings using bash syntax:

- `export JULIA_LLVM_ARGS=-print-after-all` dumps IR after each pass.
- `export JULIA_LLVM_ARGS=-debug-only=loop-vectorize` dumps LLVM `DEBUG(...)` diagnostics for loop vectorizer. If you get warnings about "Unknown command line argument", rebuild LLVM with `LLVM_ASSERTIONS = 1`.
- `export JULIA_LLVM_ARGS=-help` shows a list of available options. `export JULIA_LLVM_ARGS=-help-hidden` shows even more.
- `export JULIA_LLVM_ARGS="-fatal-warnings -print-options"` is an example how to use multiple options.

Useful `JULIA_LLVM_ARGS` parameters

- `-print-after`=PASS: prints the IR after any execution of PASS, useful for checking changes done by a pass.
- `-print-before`=PASS: prints the IR before any execution of PASS, useful for checking the input to a pass.

- `-print-changed`: prints the IR whenever a pass changes the IR, useful for narrowing down which passes are causing problems.
- `-print-(before|after)=MARKER-PASS`: the Julia pipeline ships with a number of marker passes in the pipeline, which can be used to identify where problems or optimizations are occurring. A marker pass is defined as a pass which appears once in the pipeline and performs no transformations on the IR, and is only useful for targeting print-before/print-after. Currently, the following marker passes exist in the pipeline:
  - BeforeOptimization
  - BeforeEarlySimplification
  - AfterEarlySimplification
  - BeforeEarlyOptimization
  - AfterEarlyOptimization
  - BeforeLoopOptimization
  - BeforeLICM
  - AfterLICM
  - BeforeLoopSimplification
  - AfterLoopSimplification
  - AfterLoopOptimization
  - BeforeScalarOptimization
  - AfterScalarOptimization
  - BeforeVectorization
  - AfterVectorization
  - BeforeIntrinsicLowering
  - AfterIntrinsicLowering
  - BeforeCleanup
  - AfterCleanup
  - AfterOptimization
- `-time-passes`: prints the time spent in each pass, useful for identifying which passes are taking a long time.
- `-print-module-scope`: used in conjunction with `-print-(before|after)`, gets the entire module rather than the IR unit received by the pass
- `-debug`: prints out a lot of debugging information throughout LLVM
- `-debug-only=NAME`, prints out debugging statements from files with `DEBUG_TYPE` defined to `NAME`, useful for getting additional context about a problem

Debugging LLVM transformations in isolation

On occasion, it can be useful to debug LLVM's transformations in isolation from the rest of the Julia system, e.g. because reproducing the issue inside `julia` would take too long, or because one wants to take advantage of LLVM's tooling (e.g. bugpoint).

To start with, you can install the developer tools to work with LLVM via:

```
make -C deps install-llvm-tools
```

To get unoptimized IR for the entire system image, pass the `--output-unopt-bc unopt.bc` option to the system image build process, which will output the unoptimized IR to an `unopt.bc` file. This file can then be passed to LLVM tools as usual. `libjulia` can function as an LLVM pass plugin and can be loaded into LLVM tools, to make julia-specific passes available in this environment. In addition, it exposes the `-julia` meta-pass, which runs the entire Julia pass-pipeline over the IR. As an example, to generate a system image with the old pass manager, one could do:

```

llc -o sys.o opt.bc
cc -shared -o sys.so sys.o
```

To generate a system image with the new pass manager, one could do:

```
./usr/tools/opt -load-pass-plugin=libjulia-codegen.so --passes='julia' -o opt.bc unopt.bc
./usr/tools/llc -o sys.o opt.bc
./usr/tools/cc -shared -o sys.so sys.o
```

This system image can then be loaded by `julia` as usual.

It is also possible to dump an LLVM IR module for just one Julia function, using:

```
fun, T = +, Tuple{Int,Int} # Substitute your function of interest here
optimize = false
open("plus.ll", "w") do file
    code_llvm(file, fun, T; raw=true, dump_module=true, optimize)
end
```

These files can be processed the same way as the unoptimized sysimg IR shown above, or if you want to see the LLVM IR yourself and get extra verification run, you can use

```
./usr/tools/opt -load-pass-plugin=libjulia-codegen.so --passes='julia' -S -verify-each plus.ll
```

(note on MacOS this would be `libjulia-codegen.dylib` and on Windows `libjulia-codegen.dll`)

Running the LLVM test suite

To run the llvm tests locally, you need to first install the tools, build julia, then you can run the tests:

```
make -C deps install-llvm-tools
make -j julia-src-release
make -C test/llvmpasses
```

If you want to run the individual test files directly, via the commands at the top of each test file, the first step here will have installed the tools into `./usr/tools/opt`. Then you'll want to manually replace `%s` with the name of the test file.

Improving LLVM optimizations for Julia

Improving LLVM code generation usually involves either changing Julia lowering to be more friendly to LLVM's passes, or improving a pass.

If you are planning to improve a pass, be sure to read the LLVM developer policy. The best strategy is to create a code example in a form where you can use LLVM's `opt` tool to study it and the pass of interest in isolation.

1. Create an example Julia code of interest.
2. Use `JULIA_LLVM_ARGS=-print-after-all` to dump the IR.
3. Pick out the IR at the point just before the pass of interest runs.
4. Strip the debug metadata and fix up the TBAA metadata by hand.

The last step is labor intensive. Suggestions on a better way would be appreciated.

The jlcall calling convention

Julia has a generic calling convention for unoptimized code, which looks somewhat as follows:

```
jl_value_t *any_unoptimized_call(jl_value_t *, jl_value_t **, int);
```

where the first argument is the boxed function object, the second argument is an on-stack array of arguments and the third is the number of arguments. Now, we could perform a straightforward lowering and emit an alloca for the argument array. However, this would betray the SSA nature of the uses at the call site, making optimizations (including GC root placement), significantly harder. Instead, we emit it as follows:

```
call %jl_value_t *@julia.call(jl_value_t *(*)(...) @any_unoptimized_call, %jl_value_t *%arg1,
↪  %jl_value_t *%arg2)
```

This allows us to retain the SSA-ness of the uses throughout the optimizer. GC root placement will later lower this call to the original C ABI.

GC root placement

GC root placement is done by an LLVM pass late in the pass pipeline. Doing GC root placement this late enables LLVM to make more aggressive optimizations around code that requires GC roots, as well as allowing us to reduce the number of required GC roots and GC root store operations (since LLVM doesn't understand our GC, it wouldn't otherwise know what it is and is not allowed to do with values stored to the GC frame, so it'll conservatively do very little). As an example, consider an error path

```
if some_condition()
    #= Use some variables maybe =#
    error("An error occurred")
end
```

During constant folding, LLVM may discover that the condition is always false, and can remove the basic block. However, if GC root lowering is done early, the GC root slots used in the deleted block, as well as any values kept alive in those slots only because they were used in the error path, would be kept alive by LLVM. By doing GC root lowering late, we give LLVM the license to do any of its usual optimizations (constant folding, dead code elimination, etc.), without having to worry (too much) about which values may or may not be GC tracked.

However, in order to be able to do late GC root placement, we need to be able to identify a) which pointers are GC tracked and b) all uses of such pointers. The goal of the GC placement pass is thus simple:

Minimize the number of needed GC roots/stores to them subject to the constraint that at every safepoint, any live GC-tracked pointer (i.e. for which there is a path after this point that contains a use of this pointer) is in some GC slot.

Representation

The primary difficulty is thus choosing an IR representation that allows us to identify GC-tracked pointers and their uses, even after the program has been run through the optimizer. Our design makes use of three LLVM features to achieve this:

- Custom address spaces
- Operand Bundles
- Non-integral pointers

Custom address spaces allow us to tag every point with an integer that needs to be preserved through optimizations. The compiler may not insert casts between address spaces that did not exist in the original program and it must never change the address space of a pointer on a load/store/etc operation. This allows us to annotate which pointers are GC-tracked in an optimizer-resistant way. Note that metadata would not be able to achieve the same purpose. Metadata is supposed to always be discardable without altering the semantics of the program. However, failing to identify a GC-tracked pointer alters the resulting program behavior dramatically - it'll probably crash or return wrong results. We currently use three different address spaces (their numbers are defined in `src/codegen_shared.cpp`):

- GC Tracked Pointers (currently 10): These are pointers to boxed values that may be put into a GC frame. It is loosely equivalent to a `jl_value_t*` pointer on the C side. N.B. It is illegal to ever have a pointer in this address space that may not be stored to a GC slot.

- Derived Pointers (currently 11): These are pointers that are derived from some GC tracked pointer. Uses of these pointers generate uses of the original pointer. However, they need not themselves be known to the GC. The GC root placement pass MUST always find the GC tracked pointer from which this pointer is derived and use that as the pointer to root.
- Callee Rooted Pointers (currently 12): This is a utility address space to express the notion of a callee rooted value. All values of this address space MUST be storable to a GC root (though it is possible to relax this condition in the future), but unlike the other pointers need not be rooted if passed to a call (they do still need to be rooted if they are live across another safepoint between the definition and the call).
- Pointers loaded from tracked object (currently 13): This is used by arrays, which themselves contain a pointer to the managed data. This data area is owned by the array, but is not a GC-tracked object by itself. The compiler guarantees that as long as this pointer is live, the object that this pointer was loaded from will keep being live.

Invariants

The GC root placement pass makes use of several invariants, which need to be observed by the frontend and are preserved by the optimizer.

First, only the following address space casts are allowed:

- 0->{Tracked,Derived,CalleeRooted}: It is allowable to decay an untracked pointer to any of the others. However, do note that the optimizer has broad license to not root such a value. It is never safe to have a value in address space 0 in any part of the program if it is (or is derived from) a value that requires a GC root.
- Tracked->Derived: This is the standard decay route for interior values. The placement pass will look for these to identify the base pointer for any use.
- Tracked->CalleeRooted: Addrspace CalleeRooted serves merely as a hint that a GC root is not required. However, do note that the Derived->CalleeRooted decay is prohibited, since pointers should generally be storable to a GC slot, even in this address space.

Now let us consider what constitutes a use:

- Loads whose loaded values is in one of the address spaces
- Stores of a value in one of the address spaces to a location
- Stores to a pointer in one of the address spaces
- Calls for which a value in one of the address spaces is an operand
- Calls in jlcall ABI, for which the argument array contains a value
- Return instructions.

We explicitly allow load/stores and simple calls in address spaces Tracked/Derived. Elements of jlcall argument arrays must always be in address space Tracked (it is required by the ABI that they are valid `jl_value_t*` pointers). The same is true for return instructions (though note that struct return arguments

are allowed to have any of the address spaces). The only allowable use of an address space CalleeRooted pointer is to pass it to a call (which must have an appropriately typed operand).

Further, we disallow `getelementptr` in addrspace Tracked. This is because unless the operation is a noop, the resulting pointer will not be validly storable to a GC slot and may thus not be in this address space. If such a pointer is required, it should be decayed to addrspace Derived first.

Lastly, we disallow `inttoptr`/`ptrtoint` instructions in these address spaces. Having these instructions would mean that some `i64` values are really GC tracked. This is problematic, because it breaks that stated requirement that we're able to identify GC-relevant pointers. This invariant is accomplished using the LLVM "non-integral pointers" feature, which is new in LLVM 5.0. It prohibits the optimizer from making optimizations that would introduce these operations. Note we can still insert static constants at JIT time by using `inttoptr` in address space 0 and then decaying to the appropriate address space afterwards.

Supporting `ccall`

One important aspect missing from the discussion so far is the handling of `ccall`. `ccall` has the peculiar feature that the location and scope of a use do not coincide. As an example consider:

```
A = randn(1024)
ccall(:foo, Cvoid, (Ptr{Float64},), A)
```

In lowering, the compiler will insert a conversion from the array to the pointer which drops the reference to the array value. However, we of course need to make sure that the array does stay alive while we're doing the `ccall`. To understand how this is done, lets look at a hypothetical approximate possible lowering of the above code:

```
return $(Expr(:foreigncall, Expr(:tuple, :(:foo)), Cvoid, svec(Ptr{Float64}), 0, :(:ccall),
↪  Expr(:foreigncall, Expr(:tuple, :(:jl_array_ptr)), Ptr{Float64}, svec(Any), 0, :(:ccall),
↪  :(A)), :(A)))
```

The last `:(A)`, is an extra argument list inserted during lowering that informs the code generator which Julia level values need to be kept alive for the duration of this `ccall`. We then take this information and represent it in an "operand bundle" at the IR level. An operand bundle is essentially a fake use that is attached to the call site. At the IR level, this looks like so:

```
call void inttoptr (i64 ... to void (double*)*)(double* %5) [ "jl_roots"(%jl_value_t
↪  addrspace(10)* %A) ]
```

The GC root placement pass will treat the `jl_roots` operand bundle as if it were a regular operand. However, as a final step, after the GC roots are inserted, it will drop the operand bundle to avoid confusing instruction selection.

Supporting `pointer_from_objref`

`pointer_from_objref` is special because it requires the user to take explicit control of GC rooting. By our above invariants, this function is illegal, because it performs an address space cast from 10 to 0. However, it can be useful, in certain situations, so we provide a special intrinsic:

```
declared %jl_value_t *julia.pointer_from_objref(%jl_value_t addrspace(10)*)
```

which is lowered to the corresponding address space cast after GC root lowering. Do note however that by using this intrinsic, the caller assumes all responsibility for making sure that the value in question is rooted. Further this intrinsic is not considered a use, so the GC root placement pass will not provide a GC root for the function. As a result, the external rooting must be arranged while the value is still tracked by the system. I.e. it is not valid to attempt to use the result of this operation to establish a global root - the optimizer may have already dropped the value.

Keeping values alive in the absence of uses

In certain cases it is necessary to keep an object alive, even though there is no compiler-visible use of said object. This may be case for low level code that operates on the memory-representation of an object directly or code that needs to interface with C code. In order to allow this, we provide the following intrinsics at the LLVM level:

```
token @llvm.julia.gc_preserve_begin(...)
void @llvm.julia.gc_preserve_end(token)
```

(The `llvm.` in the name is required in order to be able to use the `token` type). The semantics of these intrinsics are as follows: At any safepoint that is dominated by a `gc_preserve_begin` call, but that is not not dominated by a corresponding `gc_preserve_end` call (i.e. a call whose argument is the token returned by a `gc_preserve_begin` call), the values passed as arguments to that `gc_preserve_begin` will be kept live. Note that the `gc_preserve_begin` still counts as a regular use of those values, so the standard lifetime semantics will ensure that the values will be kept alive before entering the preserve region.

108.17 printf() and stdio in the Julia runtime

Libuv wrappers for stdio

`julia.h` defines libuv wrappers for the `stdio.h` streams:

```
uv_stream_t *JL_STDIN;
uv_stream_t *JL_STDOUT;
uv_stream_t *JL_STDERR;
```

... and corresponding output functions:

```
int jl_printf(uv_stream_t *s, const char *format, ...);
int jl_vprintf(uv_stream_t *s, const char *format, va_list args);
```

These `printf` functions are used by the `.c` files in the `src/` and `cli/` directories wherever stdio is needed to ensure that output buffering is handled in a unified way.

In special cases, like signal handlers, where the full libuv infrastructure is too heavy, `jl_safe_printf()` can be used to `write(2)` directly to `STDERR_FILENO`:

```
void jl_safe_printf(const char *str, ...);
```

Interface between JL_STD* and Julia code

`Base.stdin`, `Base.stdout` and `Base.stderr` are bound to the `JL_STD*` libuv streams defined in the runtime.

Julia's `__init__()` function (in `base/sysimg.jl`) calls `reinit_stdio()` (in `base/stream.jl`) to create Julia objects for `Base.stdin`, `Base.stdout` and `Base.stderr`.

`reinit_stdio()` uses `ccall` to retrieve pointers to `JL_STD*` and calls `jl_uv_handle_type()` to inspect the type of each stream. It then creates a Julia `Base.IOStream`, `Base.TTY` or `Base.PipeEndpoint` object to represent each stream, e.g.:

```
$ julia -e 'println(typeof((stdin, stdout, stderr)))'
Tuple{Base.TTY,Base.TTY,Base.TTY}

$ julia -e 'println(typeof((stdin, stdout, stderr)))' < /dev/null 2>/dev/null
Tuple{IOStream,Base.TTY,IOStream}

$ echo hello | julia -e 'println(typeof((stdin, stdout, stderr)))' | cat
Tuple{Base.PipeEndpoint,Base.PipeEndpoint,Base.TTY}
```

The `Base.read` and `Base.write` methods for these streams use `ccall` to call libuv wrappers in `src/jl_uv.c`, e.g.:

```
stream.jl: function write(s::IO, p::Ptr, nb::Integer)
               -> ccall(:jl_uv_write, ...)
  jl_uv.c:          -> int jl_uv_write(uv_stream_t *stream, ...)
                        -> uv_write(uvw, stream, buf, ...)
```

printf() during initialization

The libuv streams relied upon by `jl_printf()` etc., are not available until midway through initialization of the runtime (see `init.c`, `init_stdio()`). Error messages or warnings that need to be printed before this are routed to the standard C library `fwrite()` function by the following mechanism:

In `sys.c`, the `JL_STD*` stream pointers are statically initialized to integer constants: `STD*_FILENO (0, 1 and 2)`. In `jl_uv.c` the `jl_uv_puts()` function checks its `uv_stream_t* stream` argument and calls `fwrite()` if stream is set to `STDOUT_FILENO` or `STDERR_FILENO`.

This allows for uniform use of `jl_printf()` throughout the runtime regardless of whether or not any particular piece of code is reachable before initialization is complete.

Legacy `ios.c` library

The `src/support/ios.c` library is inherited from femtolisp. It provides cross-platform buffered file IO and in-memory temporary buffers.

`ios.c` is still used by:

- `src/flisp/*.c`
- `src/dump.c` – for serialization file IO and for memory buffers.

- `src/staticdata.c` – for serialization file IO and for memory buffers.
- `base/iostream.jl` – for file IO (see `base/fs.jl` for libuv equivalent).

Use of `ios.c` in these modules is mostly self-contained and separated from the libuv I/O system. However, there is one place where femtolisp calls through to `jl_printf()` with a legacy `ios_t` stream.

There is a hack in `ios.h` that makes the `ios_t.bm` field line up with the `uv_stream_t.type` and ensures that the values used for `ios_t.bm` to not overlap with valid `UV_HANDLE_TYPE` values. This allows `uv_stream_t` pointers to point to `ios_t` streams.

This is needed because `jl_printf()` caller `jl_static_show()` is passed an `ios_t` stream by femtolisp's `fl_print()` function. Julia's `jl_uv_puts()` function has special handling for this:

```
if (stream->type > UV_HANDLE_TYPE_MAX) {
    return ios_write((ios_t*)stream, str, n);
}
```

108.18 Bounds checking

Like many modern programming languages, Julia uses bounds checking to ensure program safety when accessing arrays. In tight inner loops or other performance critical situations, you may wish to skip these bounds checks to improve runtime performance. For instance, in order to emit vectorized (SIMD) instructions, your loop body cannot contain branches, and thus cannot contain bounds checks. Consequently, Julia includes an `@inbounds(...)` macro to tell the compiler to skip such bounds checks within the given block. User-defined array types can use the `@boundscheck(...)` macro to achieve context-sensitive code selection.

Eliding bounds checks

The `@boundscheck(...)` macro marks blocks of code that perform bounds checking. When such blocks are inlined into an `@inbounds(...)` block, the compiler may remove these blocks. The compiler removes the `@boundscheck` block *only if it is inlined* into the calling function. For example, you might write the method `sum` as:

```
function sum(A::AbstractArray)
    r = zero(eltype(A))
    for i in eachindex(A)
        @inbounds r += A[i]
    end
    return r
end
```

With a custom array-like type `MyArray` having:

```
@inline getindex(A::MyArray, i::Real) = (@boundscheck checkbounds(A, i); A.data[to_index(i)])
```

Then when `getindex` is inlined into `sum`, the call to `checkbounds(A, i)` will be elided. If your function contains multiple layers of inlining, only `@boundscheck` blocks at most one level of inlining deeper are eliminated. The rule prevents unintended changes in program behavior from code further up the stack.

Caution!

It is easy to accidentally expose unsafe operations with `@inbounds`. You might be tempted to write the above example as

```
function sum(A::AbstractArray)
    r = zero(eltype(A))
    for i in 1:length(A)
        @inbounds r += A[i]
    end
    return r
end
```

Which quietly assumes 1-based indexing and therefore exposes unsafe memory access when used with `OffsetArrays`:

```
julia> using OffsetArrays

julia> sum(OffsetArray([1, 2, 3], -10))
9164911648 # inconsistent results or segfault
```

While the original source of the error here is `1:length(A)`, the use of `@inbounds` increases the consequences from a bounds error to a less easily caught and debugged unsafe memory access. It is often difficult or impossible to prove that a method which uses `@inbounds` is safe, so one must weigh the benefits of performance improvements against the risk of segfaults and silent misbehavior, especially in public facing APIs.

Propagating inbounds

There may be certain scenarios where for code-organization reasons you want more than one layer between the `@inbounds` and `@boundscheck` declarations. For instance, the default `getindex` methods have the chain `getindex(A::AbstractArray, i::Real)` calls `getindex(IndexStyle(A), A, i)` calls `_getindex(::IndexLinear, A, i)`.

To override the "one layer of inlining" rule, a function may be marked with `Base.@propagate_inbounds` to propagate an inbounds context (or out of bounds context) through one additional layer of inlining.

The bounds checking call hierarchy

The overall hierarchy is:

- `checkbounds(A, I...)` which calls
  - `checkbounds(Bool, A, I...)` which calls
    - `checkbounds_indices(Bool, axes(A), I)` which recursively calls
      - `checkindex` for each dimension

Here A is the array, and I contains the "requested" indices. axes(A) returns a tuple of "permitted" indices of A.

checkbounds(A, I...) throws an error if the indices are invalid, whereas checkbounds(Bool, A, I...) returns false in that circumstance. checkbounds_indices discards any information about the array other than its axes tuple, and performs a pure indices-vs-indices comparison: this allows relatively few compiled methods to serve a huge variety of array types. Indices are specified as tuples, and are usually compared in a 1-1 fashion with individual dimensions handled by calling another important function, checkindex: typically,

```
checkbounds_indices(Bool, (IA1, IA...), (I1, I...)) = checkindex(Bool, IA1, I1) &
                                                      checkbounds_indices(Bool, IA, I)
```

so checkindex checks a single dimension. All of these functions, including the unexported checkbounds_indices have docstrings accessible with ? .

If you have to customize bounds checking for a specific array type, you should specialize checkbounds(Bool, A, I...). However, in most cases you should be able to rely on checkbounds_indices as long as you supply useful axes for your array type.

If you have novel index types, first consider specializing checkindex, which handles a single index for a particular dimension of an array. If you have a custom multidimensional index type (similar to CartesianIndex), then you may have to consider specializing checkbounds_indices.

Note this hierarchy has been designed to reduce the likelihood of method ambiguities. We try to make checkbounds the place to specialize on array type, and try to avoid specializations on index types; conversely, checkindex is intended to be specialized only on index type (especially, the last argument).

Emit bounds checks

Julia can be launched with --check-bounds={yes|no|auto} to emit bounds checks always, never, or respect @inbounds declarations.

108.19 Proper maintenance and care of multi-threading locks

The following strategies are used to ensure that the code is dead-lock free (generally by addressing the 4th Coffman condition: circular wait).

1. structure code such that only one lock will need to be acquired at a time
2. always acquire shared locks in the same order, as given by the table below
3. avoid constructs that expect to need unrestricted recursion

Types of locks

uv_mutex_t (or std::mutex) is a wrapper around platform-specific locks (pthread_mutex_t on Unix, CRITICAL_SECTION on Windows). It may cause the current OS thread to block, is not reentrant, and is not a safepoint.

jl_mutex_t is a reentrant spinlock. jl_mutex_ts acquired in a try block will be unlocked when we leave the block, either by reaching the end or catching an exception. JL_LOCK/JL_UNLOCK are safepoints, while JL_LOCK_NOGC/JL_UNLOCK_NOGC are not. jl_mutex_t must not be held across task switches.

Lock hierarchy

Below are all of the locks that exist in the system and the mechanisms for using them that avoid the potential for deadlocks (no Ostrich algorithm allowed here). Except in the special cases where a rule for avoiding deadlock is given, no lock of a lower level may acquire a lock at a higher level.

Level 1

No other lock may be acquired when one of these locks is held. As a result, the code must not do any allocation or hit any safepoints. Note that there are safepoints when doing allocation, enabling/disabling GC, entering/restoring exception frames, and taking/releasing locks.

- `safepoint_lock` (`uv_mutex_t`)

  Danger

  This lock is acquired implicitly by `JL_LOCK` and `JL_UNLOCK`. Use the `_NOGC` variants to avoid that for level 1 locks.

- `shared_map_lock.mtx` (`uv_mutex_t`)
- `finalizers_lock` (`jl_mutex_t`)
- `gc_pages_lock` (`uv_mutex_t`)
- `gc_perm_lock` (`uv_mutex_t`)
- `gc_queue_observer_lock` (`uv_mutex_t`)
- `gc_threads_lock` (`uv_mutex_t`)
- `flisp_lock` (`uv_mutex_t`)

  Note

  flisp itself is already threadsafe; this lock only protects the `jl_ast_context_list_t` pool. Likewise, the `ResourcePool<?>::mutexes` just protect the associated resource pool.

- `ResourcePool<?>.mutex` (`std::mutex`)
- `RLST_mutex` (`std::mutex`)
- `llvm_printing_mutex` (`std::mutex`)
- `jl_locked_stream.mutex` (`std::mutex`)
- `debuginfo_asyncsafe` (`uv_rwlock_t`) (can still acquire `jl_in_stackwalk` (`uv_mutex_t`, Win32 only))
- `profile_show_peek_cond_lock` (`jl_mutex_t`)
- `trampoline_lock` (`uv_mutex_t`)
- `bt_data_prof_lock` (`uv_mutex_t`)
- `jl_ptls_t.sleep_lock` (`uv_mutex_t`)

- `tls_lock` (`uv_mutex_t`)
- `page_profile_lock` (`uv_mutex_t`)
- `symtab_lock` (`uv_mutex_t`)
- `engine_lock` (`std::mutex`)

Level 2

- `global_roots_lock`
- `jl_module_t.lock`
- `newly_inferred_mutex`
- `JLDebuginfoPlugin.PluginMutex` (`std::mutex`)
- `precompile_field_replace_lock`
- `live_tasks_lock` (`uv_mutex_t`)
- `heapsnapshot_lock`
- `jitlock`
- `jl_safepoint_suspend_all_threads` and `jl_safepoint_resume_all_threads`

  Note

  Inside a region protected by these functions, all other threads are blocked inside a safepoint. It is unsafe to take locks that may safepoint in this region.

Level 3

- `jl_method_t.writelock`
- `typecache_lock`
- `libmap_lock`

Level 4

- `jl_methcache_t.writelock`

Level 5

- `jl_methtable_t.writelock`

Level 6

No Julia code may be called while holding a lock above this point.

- `world_counter_lock`
- `jl_typeinf_lock`

Level 7

- `JuliaOJIT::EmissionMutex` (`std::recursive_mutex`)
- `jl_modules_mutex`
- `jl_uv_mutex` (known as `iolock` from Julia)

  Danger

  Doing any I/O (for example, printing warning messages or debug information) while holding any other lock listed above may result in pernicious and hard-to-find deadlocks.

- Individual `ThreadSynchronizer` locks

  Danger

  This may continue to be held after releasing the iolock, or acquired without it, but be very careful to never attempt to acquire the iolock while holding it.

- `Libdl.LazyLibrary.lock` (`ReentrantLock`)
- `orc::ThreadSafeContext`
- `cfun_lock`

Level 8

- `precomp_statement_out_lock`
- `dispatch_statement_out_lock`

Exceptions to the lock hierarchy

Ordinarily, it is forbidden to acquire locks of equal level to a lock already held. In these specific cases we use a special protocol for acquiring locks at the same level:

- `jl_method_t.writelock`

  Invalidation acquires the lock for every method during its depth-first search for backedges. To avoid deadlocks, we must already hold `world_counter_lock` before acquiring multiple `jl_method_t.writelocks`.

Broken locks

The following locks are broken:

- `loading.jl`: `require` and `register_root_module`

  This file potentially has numerous problems. (fix: needs locks)

Updates to the world counter

Thanks to the world age mechanism, Julia can allow the replacement of both methods and bindings, yet remain amenable to optimization. Every compiled `CodeInstance` has a range of valid world ages; we could conservatively assume all CIs are stale after a world age increment. However, to avoid spurious recompilation, we track dependencies, called "edges", while maintaining the following invariant:

For every published `CodeInstance`, either:

- `min_world` and `max_world` are finite, and the CI is valid for every world in that range.
- `max_world` is ∞ (`-1`), and this CI is ready for invalidation, meaning for every forward edge:
  - If the edge is a `CodeInstance` that is invoked or inlined into this CI, the edge's `MethodInstance` backedge array has an entry pointing back.
  - If the edge is a `Binding`:
    * If the binding is in another module, it has an entry for this CI in its backedges array.
    * If the binding is in the same module, the `Method` for this CI is in the module's `scanned_methods` array.

For example, the following code replaces a constant in another module, causing a chain of invalidations:

```
const c1 = 1
module M const c2 = 2 end
f() = getfield(M, :c2)
g() = f() + c1

g()                   # compile g

@eval M const c2 = 3  # invalidate f, g
g()                   # recompile g
```

After compiling the two versions of `g()`, the global cache looks like this:

The maximum world age, `jl_world_counter`, is protected by the `world_counter_lock`. Julia uses a form of optimistic concurrency control to allow type inference without holding `world_counter_lock`.

Publishing a new method or binding follows these steps:

- Acquire `world_counter_lock`.
- Relaxed-load `jl_world_counter` and let `new_world = jl_world_counter + 1`.
- Publish the new binding partitions or method table entries with world range `[new_world, ∞)`. This step is described in the section on the lock free data structures.
- Release-store `new_world` to `jl_world_counter`.
- Release `world_counter_lock`.

Type inference proceeds like so:

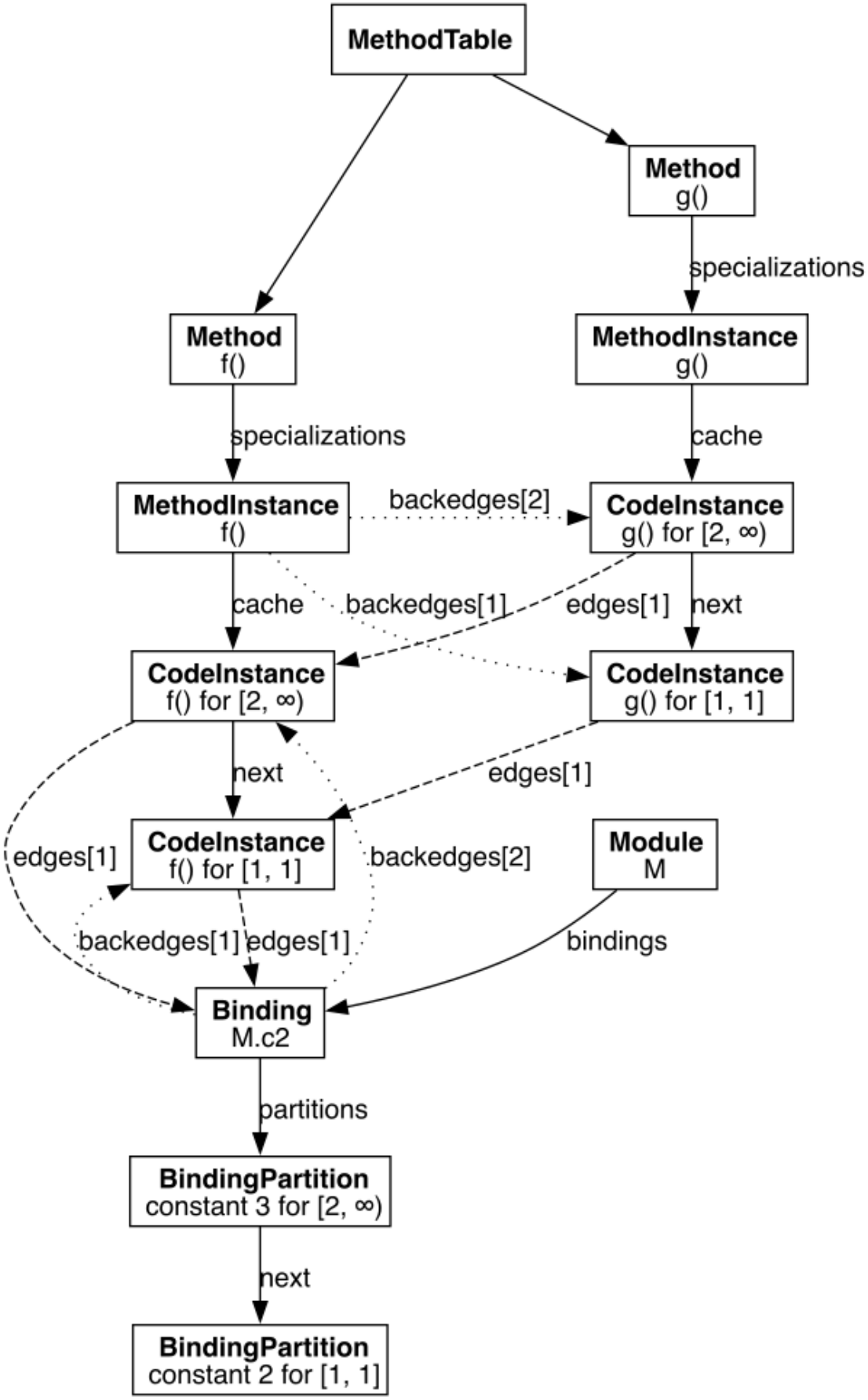


Figure 108.2: Global cache state after invalidation

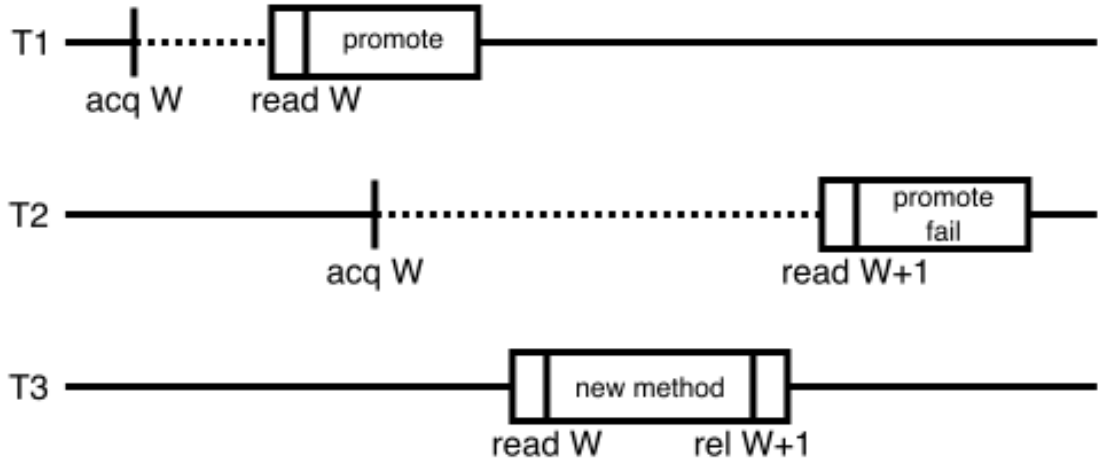


Figure 108.3: Two threads doing type inference while another adds a method

- Acquire-load `jl_world_counter` (call this `validation_world`).
- Perform type inference in that world, reading the bindings and method table in that world using the lock-free data structures.
- Store back edges for every inferred `CodeInstance`:
  - For non-local bindings, this acquires the binding's module's lock.
  - For CIs, this acquires the method's lock.
- Acquire `world_counter_lock`.
- Relaxed-load `jl_world_counter` and compare it to `validation_world`:
  - If it is different, leave the valid world ranges for the inferred CIs unchanged.
  - If it is unchanged, our optimism was rewarded. We can promote all the inferred CIs valid in `validation_world` to `[validation_world, ∞)` and rely on the backedges for invalidation.
- Release `world_counter_lock`.

In the above diagram, threads 1 and 2 are doing type inference (the dotted line), while thread 3 is activating a new method. The solid boxes represent critical sections where the `world_counter_lock` is held. `acq`, `rel`, and `read`, are acquire loads, release stores, and relaxed loads respectively.

T1 promotes its CI in time, but T2 takes too long, blocking on `world_counter_lock` until T3 has finished publishing the new method and incrementing the world counter. It reads `W+1` and fails to promote its CI, leaving it with a maximum world of `W`.

Lock free data structures

TODO

108.20 Task scheduler wakeups

This page documents the sleep/wake handshake used by Julia's task scheduler and, in particular, why `jl_wakeup_threadpool` may wake only a *single* worker per multiqueue insert instead of broadcasting to every thread.

The problem

Every `@spawn` that lands in the multiqueue used to call `jl_wakeup_thread(-1)`, which loops over *all* threads and issues a `uv_cond_signal` to each sleeping one. On an oversubscribed machine a burst of spawns therefore produces an $O(nthreads)$ "wake storm" per insert (JuliaLang/julia#61820, JuliaLang/julia#50425): every idle worker is woken, races for the one new task, and all but one immediately go back to sleep.

`jl_wakeup_threadpool(tpid)` replaces that broadcast. A multiqueue insert wakes at most one sleeping worker *in the task's pool*. A burst of N inserts wakes up to N workers. There is no peer-to-peer cascade: a woken worker simply pops its task and runs it.

Why waking one worker is enough

Correctness rests on the same store-buffering fence (`[^store_buffering_1]` in `src/scheduler.c`) that the broadcast path relied on. The relevant per-thread invariant is:

> A worker is observed `sleeping` by the waker, **or** it has not yet committed to its sleep transition and is therefore guaranteed to re-check its queue and find the freshly-inserted task.

The consumer's sleep transition in `jl_task_get_next` is, in order:

1. publish `sleep_check_state = sleeping`,
2. `jl_fence()`,
3. re-check the queue (`check_empty`); abort the sleep if work appeared,
4. decrement `n_threads_running`,
5. park on the condition variable while `may_sleep` holds.

The enqueuer fences after the insert and then inspects each candidate's `sleep_check_state`. Because the consumer publishes `sleeping` in step 1 — before it re-checks the queue in step 3 — the enqueuer either sees `sleeping` (and wakes the worker) or the worker will itself observe the new task in step 3. Either way the task is serviced.

Why the wake must *not* be skipped using `n_threads_running`

A tempting optimization is to skip the scan entirely when `n_threads_running >= jl_n_threads` ("everything is already running, nothing is parked"). This is **unsound**, for two independent reasons:

1. **The count lags the per-thread state.** `n_threads_running` is decremented in step 4, *after* the consumer has already re-checked its queue in step 3. In the window between steps 1 and 4 a worker has published `sleeping` and may have already read its queue as empty, yet is still counted as running. An enqueuer that consults the count therefore reads stale "all running" information and skips a wake the worker actually needs.

2. **The count is global but the wake is pool-local.** Busy workers in one pool keep `n_threads_running` high while another pool is entirely parked. A cross-pool insert (e.g. spawning an `:interactive` task from a `:default` worker) would then be dropped, stranding the task indefinitely.

Scanning `sleep_check_state` avoids both problems: that flag is set in step 1, *before* the re-check, so a scan never misses a worker in the danger window, and it is inspected per-pool.

TLA+ model

The directory `scheduler-wakeup/` contains a small TLA+ model of this handshake that makes the argument above machine-checkable.

- `SchedulerWake.tla` models the sleep transition as the same discrete steps listed above, so the model checker explores every interleaving of that window against an enqueuer that wakes one worker by scanning `sleep_check_state`.
- `MCFixed` — a concrete instance (two workers sharing a pool, plus a cross-pool producer). TLC explores the full state space with no deadlock and no `NoLostWakeup` violation.

To reproduce, with `tla2tools.jar`:

```
cd doc/src/devdocs/scheduler-wakeup
java -cp tla2tools.jar tlc2.TLC -config MCFixed.cfg MCFixed.tla
```

Toggling the model back to the unsound `n_threads_running` short-circuit (by making `Wakeup` return early when `nrun >= N`) makes TLC report a `NoLostWakeup` violation, confirming the danger window described above is real.

108.21 Arrays with custom indices

Conventionally, Julia's arrays are indexed starting at 1, whereas some other languages start numbering at 0, and yet others (e.g., Fortran) allow you to specify arbitrary starting indices. While there is much merit in picking a standard (i.e., 1 for Julia), there are some algorithms which simplify considerably if you can index outside the range `1:size(A,d)` (and not just `0:size(A,d)-1`, either). To facilitate such computations, Julia supports arrays with arbitrary indices.

The purpose of this page is to address the question, "what do I have to do to support such arrays in my own code?" First, let's address the simplest case: if you know that your code will never need to handle arrays with unconventional indexing, hopefully the answer is "nothing." Old code, on conventional arrays, should function essentially without alteration as long as it was using the exported interfaces of Julia. If you find it more convenient to just force your users to supply traditional arrays where indexing starts at one, you can add

```
Base.require_one_based_indexing(arrays...)
```

where `arrays...` is a list of the array objects that you wish to check for anything that violates 1-based indexing.

Generalizing existing code

As an overview, the steps are:

- replace many uses of `size` with `axes`
- replace `1:length(A)` with `eachindex(A)`, or in some cases `LinearIndices(A)`
- replace explicit allocations like `Array{Int}(undef, size(B))` with `similar(Array{Int}, axes(B))`

These are described in more detail below.

Things to watch out for

Because unconventional indexing breaks many people's assumptions that all arrays start indexing with 1, there is always the chance that using such arrays will trigger errors. The most frustrating bugs would be incorrect results or segfaults (total crashes of Julia). For example, consider the following function:

```
function mycopy!(dest::AbstractVector, src::AbstractVector)
    length(dest) == length(src) || throw(DimensionMismatch("vectors must match"))
    # OK, now we're safe to use @inbounds, right? (not anymore!)
    for i = 1:length(src)
        @inbounds dest[i] = src[i]
    end
    dest
end
```

This code implicitly assumes that vectors are indexed from 1; if `dest` starts at a different index than `src`, there is a chance that this code would trigger a segfault. (If you do get segfaults, to help locate the cause try running julia with the option `--check-bounds=yes`.)

Using axes for bounds checks and loop iteration

`axes(A)` (reminiscent of `size(A)`) returns a tuple of `AbstractUnitRange{<:Integer}` objects, specifying the range of valid indices along each dimension of `A`. When `A` has unconventional indexing, the ranges may not start at 1. If you just want the range for a particular dimension `d`, there is `axes(A, d)`.

Base implements a custom range type, `OneTo`, where `OneTo(n)` means the same thing as `1:n` but in a form that guarantees (via the type system) that the lower index is 1. For any new `AbstractArray` type, this is the default returned by `axes`, and it indicates that this array type uses "conventional" 1-based indexing.

For bounds checking, note that there are dedicated functions `checkbounds` and `checkindex` which can sometimes simplify such tests.

Linear indexing (`LinearIndices`)

Some algorithms are most conveniently (or efficiently) written in terms of a single linear index, `A[i]` even if `A` is multi-dimensional. Regardless of the array's native indices, linear indices always range from `1:length(A)`. However, this raises an ambiguity for one-dimensional arrays (a.k.a., `AbstractVector`): does `v[i]` mean linear indexing , or Cartesian indexing with the array's native indices?

For this reason, your best option may be to iterate over the array with `eachindex(A)`, or, if you require the indices to be sequential integers, to get the index range by calling `LinearIndices(A)`. This will return `axes(A, 1)` if A is an AbstractVector, and the equivalent of `1:length(A)` otherwise.

By this definition, 1-dimensional arrays always use Cartesian indexing with the array's native indices. To help enforce this, it's worth noting that the index conversion functions will throw an error if shape indicates a 1-dimensional array with unconventional indexing (i.e., is a `Tuple{UnitRange}` rather than a tuple of `OneTo`). For arrays with conventional indexing, these functions continue to work the same as always.

Using `axes` and `LinearIndices`, here is one way you could rewrite `mycopy!`:

```
function mycopy!(dest::AbstractVector, src::AbstractVector)
    axes(dest) == axes(src) || throw(DimensionMismatch("vectors must match"))
    for i in LinearIndices(src)
        @inbounds dest[i] = src[i]
    end
    dest
end
```

Allocating storage using generalizations of `similar`

Storage is often allocated with `Array{Int}(undef, dims)` or `similar(A, args...)`. When the result needs to match the indices of some other array, this may not always suffice. The generic replacement for such patterns is to use `similar(storagetype, shape)`. `storagetype` indicates the kind of underlying "conventional" behavior you'd like, e.g., `Array{Int}` or `BitArray` or even `dims->zeros(Float32, dims)` (which would allocate an all-zeros array). `shape` is a tuple of `Integer` or `AbstractUnitRange` values, specifying the indices that you want the result to use. Note that a convenient way of producing an all-zeros array that matches the indices of A is simply `zeros(A)`.

Let's walk through a couple of explicit examples. First, if A has conventional indices, then `similar(Array{Int}, axes(A))` would end up calling `Array{Int}(undef, size(A))`, and thus return an array. If A is an `AbstractArray` type with unconventional indexing, then `similar(Array{Int}, axes(A))` should return something that "behaves like" an `Array{Int}` but with a shape (including indices) that matches A. (The most obvious implementation is to allocate an `Array{Int}(undef, size(A))` and then "wrap" it in a type that shifts the indices.)

Note also that `similar(Array{Int}, (axes(A, 2),))` would allocate an `AbstractVector{Int}` (i.e., 1-dimensional array) that matches the indices of the columns of A.

Writing custom array types with non-1 indexing

Most of the methods you'll need to define are standard for any `AbstractArray` type, see Abstract Arrays. This page focuses on the steps needed to define unconventional indexing.

Custom `AbstractUnitRange` types

If you're writing a non-1 indexed array type, you will want to specialize `axes` so it returns a `UnitRange`, or (perhaps better) a custom `AbstractUnitRange`. The advantage of a custom type is that it "signals" the allocation type for functions like `similar`. If we're writing an array type for which indexing will start at 0, we likely want to begin by creating a new `AbstractUnitRange`, `ZeroRange`, where `ZeroRange(n)` is equivalent to `0:n-1`.

In general, you should probably *not* export ZeroRange from your package: there may be other packages that implement their own ZeroRange, and having multiple distinct ZeroRange types is (perhaps counterintuitively) an advantage: `ModuleA.ZeroRange` indicates that `similar` should create a `ModuleA.ZeroArray`, whereas `ModuleB.ZeroRange` indicates a `ModuleB.ZeroArray` type. This design allows peaceful coexistence among many different custom array types.

Note that the Julia package CustomUnitRanges.jl can sometimes be used to avoid the need to write your own ZeroRange type.

Specializing axes

Once you have your `AbstractUnitRange` type, then specialize `axes`:

```
Base.axes(A::ZeroArray) = map(n->ZeroRange(n), A.size)
```

where here we imagine that ZeroArray has a field called `size` (there would be other ways to implement this).

In some cases, the fallback definition for `axes(A, d)`:

```
axes(A::AbstractArray{T,N}, d) where {T,N} = d <= N ? axes(A)[d] : OneTo(1)
```

may not be what you want: you may need to specialize it to return something other than `OneTo(1)` when `d > ndims(A)`. Likewise, in Base there is a dedicated function `axes1` which is equivalent to `axes(A, 1)` but which avoids checking (at runtime) whether `ndims(A) > 0`. (This is purely a performance optimization.) It is defined as:

```
axes1(A::AbstractArray{T,0}) where {T} = OneTo(1)
axes1(A::AbstractArray) = axes(A)[1]
```

If the first of these (the zero-dimensional case) is problematic for your custom array type, be sure to specialize it appropriately.

Specializing `similar`

Given your custom ZeroRange type, then you should also add the following two specializations for `similar`:

```
function Base.similar(A::AbstractArray, T::Type, shape::Tuple{ZeroRange,Vararg{ZeroRange}})
    # body
end

function Base.similar(f::Union{Function,DataType}, shape::Tuple{ZeroRange,Vararg{ZeroRange}})
    # body
end
```

Both of these should allocate your custom array type.

Specializing reshape

Optionally, define a method

```
Base.reshape(A::AbstractArray, shape::Tuple{ZeroRange,Vararg{ZeroRange}}) = ...
```

and you can reshape an array so that the result has custom indices.

For objects that mimic AbstractArray but are not subtypes

has_offset_axes depends on having axes defined for the objects you call it on. If there is some reason you don't have an axes method defined for your object, consider defining a method

```
Base.has_offset_axes(obj::MyNon1IndexedArraylikeObject) = true
```

This will allow code that assumes 1-based indexing to detect a problem and throw a helpful error, rather than returning incorrect results or segfaulting julia.

Catching errors

If your new array type triggers errors in other code, one helpful debugging step can be to comment out `@boundscheck` in your `getindex` and `setindex!` implementation. This will ensure that every element access checks bounds. Or, restart julia with `--check-bounds=yes`.

In some cases it may also be helpful to temporarily disable `size` and `length` for your new array type, since code that makes incorrect assumptions frequently uses these functions.

108.22 Module loading

`Base.require` is responsible for loading modules and it also manages the precompilation cache. It is the implementation of the `import` statement.

Experimental features

The features below are experimental and not part of the stable Julia API. Before building upon them inform yourself about the current thinking and whether they might change soon.

Package loading callbacks

It is possible to listen to the packages loaded by `Base.require`, by registering a callback.

```
loaded_packages = Base.PkgId[]
callback = (pkg::Base.PkgId) -> push!(loaded_packages, pkg)
push!(Base.package_callbacks, callback)
```

Using this would look something like:

```
julia> using Example

julia> loaded_packages
1-element Vector{Base.PkgId}:
 Example [7876af07-990d-54b4-ab0e-23690620f79a]
```

108.23 Inference

How inference works

In Julia compiler, "type inference" refers to the process of deducing the types of later values from the types of input values. Julia's approach to inference has been described in the blog posts below:

1. Shows a simplified implementation of the data-flow analysis algorithm, that Julia's type inference routine is based on.
2. Gives a high level view of inference with a focus on its inter-procedural convergence guarantee.
3. Explains a refinement on the algorithm introduced in 2.

Debugging compiler.jl

You can start a Julia session, edit `compiler/*.jl` (for example to insert `print` statements), and then replace `Core.Compiler` in your running session by navigating to `base` and executing `include("compiler/compiler.jl")`. This trick typically leads to much faster development than if you rebuild Julia for each change.

Alternatively, you can use the Revise.jl package to track the compiler changes by using the command `Revise.track(Core.Compiler)` at the beginning of your Julia session. As explained in the Revise documentation, the modifications to the compiler will be reflected when the modified files are saved.

A convenient entry point into inference is `typeinf_code`. Here's a demo running inference on `convert(Int, UInt(1))`:

```
# Get the method
atypes = Tuple{Type{Int}, UInt}  # argument types
mths = methods(convert, atypes)  # worth checking that there is only one
m = first(mths)

# Create variables needed to call `typeinf_code`
interp = Core.Compiler.NativeInterpreter()
sparams = Core.svec()      # this particular method doesn't have type-parameters
run_optimizer = true       # run all inference optimizations
types = Tuple{typeof(convert), atypes.parameters...} # Tuple{typeof(convert), Type{Int}, UInt}
Core.Compiler.typeinf_code(interp, m, types, sparams, run_optimizer)
```

If your debugging adventures require a `MethodInstance`, you can look it up by calling `Core.Compiler.specialize_method` using many of the variables above. A `CodeInfo` object may be obtained with

```
# Returns the CodeInfo object for `convert(Int, ::UInt)`:
ci = (@code_typed convert(Int, UInt(1)))[1]
```

The inlining algorithm (`inline_worthy`)

Much of the hardest work for inlining runs in `ssa_inlining_pass!`. However, if your question is "why didn't my function inline?" then you will most likely be interested in `inline_worthy`, which makes a decision to inline the function call or not.

`inline_worthy` implements a cost-model, where "cheap" functions get inlined; more specifically, we inline functions if their anticipated run-time is not large compared to the time it would take to issue a call to them if

they were not inlined. The cost-model is extremely simple and ignores many important details: for example, all `for` loops are analyzed as if they will be executed once, and the cost of an `if...else...end` includes the summed cost of all branches. It's also worth acknowledging that we currently lack a suite of functions suitable for testing how well the cost model predicts the actual run-time cost, although BaseBenchmarks provides a great deal of indirect information about the successes and failures of any modification to the inlining algorithm.

The foundation of the cost-model is a lookup table, implemented in `add_tfunc` and its callers, that assigns an estimated cost (measured in CPU cycles) to each of Julia's intrinsic functions. These costs are based on standard ranges for common architectures (see Agner Fog's analysis for more detail).

We supplement this low-level lookup table with a number of special cases. For example, an `:invoke` expression (a call for which all input and output types were inferred in advance) is assigned a fixed cost (currently 20 cycles). In contrast, a `:call` expression, for functions other than intrinsics/builtins, indicates that the call will require dynamic dispatch, in which case we assign a cost set by `Params.inline_nonleaf_penalty` (currently set at `1000`). Note that this is not a "first-principles" estimate of the raw cost of dynamic dispatch, but a mere heuristic indicating that dynamic dispatch is extremely expensive.

Each statement gets analyzed for its total cost in a function called `statement_cost`. You can display the cost associated with each statement as follows:

```
julia> Base.print_statement_costs(stdout, map, (typeof(sqrt), Tuple{Int},)) # map(sqrt, (2,))
map(f, t::Tuple{Any}) @ Base tuple.jl:358
  0 1 ─ %1  = $(Expr(:boundscheck, true))::Bool
  0 │   %2  =   builtin Base.getfield(_3, 1, %1)::Int64
  1 │   %3  = intrinsic Base.sitofp(Float64, %2)::Float64
  0 │   %4  = intrinsic Base.lt_float(%3, 0.0)::Bool
  0 └──       goto #3 if not %4
  0 2 ─          invoke Base.Math.throw_complex_domainerror(:sqrt::Symbol, %3::Float64)::Union{}
  0 └──       unreachable
 20 3 ─ %8  = intrinsic Base.Math.sqrt_llvm(%3)::Float64
  0 └──       goto #4
  0 4 ─       goto #5
  0 5 ─ %11 =   builtin Core.tuple(%8)::Tuple{Float64}
  0 └──       return %11
```

The line costs are in the left column. This includes the consequences of inlining and other forms of optimization.

108.24 Julia SSA-form IR

Julia uses a static single assignment intermediate representation (SSA IR) to perform optimization. This IR is different from LLVM IR, and unique to Julia. It allows for Julia specific optimizations.

1. Basic blocks (regions with no control flow) are explicitly annotated.
2. if/else and loops are turned into `goto` statements.
3. lines with multiple operations are split into multiple lines by introducing variables.

For example the following Julia code:

```
function foo(x)
    y = sin(x)
    if x > 5.0
        y = y + cos(x)
    end
    return exp(2) + y
end
```

when called with a `Float64` argument is translated into:

```
using InteractiveUtils
@code_typed foo(1.0)
```

```
CodeInfo(
1 — %1 = invoke Main.sin(x::Float64)::Float64
│   %2 = Base.lt_float(x, 5.0)::Bool
└──      goto #3 if not %2
2 — %4 = invoke Main.cos(x::Float64)::Float64
└── %5 = Base.add_float(%1, %4)::Float64
3 ┄ %6 = φ (#2 => %5, #1 => %1)::Float64
│   %7 = Base.add_float(7.38905609893065, %6)::Float64
└──      return %7
) => Float64
```

In this example, we can see all of these changes.

1. The first basic block is everything in

```
1 — %1 = invoke Main.sin(x::Float64)::Float64
│   %2 = Base.lt_float(x, 5.0)::Bool
└──      goto #3 if not %2
```

1. The `if` statement is translated into `goto #3 if not %2` which goes to the 3rd basic block if x>5 isn't met and otherwise goes to the second basic block.
2. `%2` is an SSA value introduced to represent `x > 5`.

Background

Beginning in Julia 0.7, parts of the compiler use a new SSA-form intermediate representation (IR). Historically, the compiler would directly generate LLVM IR from a lowered form of the Julia AST. This form had most syntactic abstractions removed, but still looked a lot like an abstract syntax tree. Over time, in order to facilitate optimizations, SSA values were introduced to this IR and the IR was linearized (i.e. turned into a form where function arguments could only be SSA values or constants). However, non-SSA values (slots) remained in the IR due to the lack of Phi nodes in the IR (necessary for back-edges and re-merging of conditional control flow). This negated much of the usefulness of SSA form representation when performing middle end optimizations. Some heroic effort was put into making these optimizations work without a complete SSA form representation, but the lack of such a representation ultimately proved prohibitive.

Categories of IR nodes

The SSA IR representation has four categories of IR nodes: Phi, Pi, PhiC, and Upsilon nodes (the latter two are only used for exception handling).

Phi nodes and Pi nodes

Phi nodes are part of generic SSA abstraction (see the link above if you're not familiar with the concept). In the Julia IR, these nodes are represented as:

```
struct PhiNode
    edges::Vector{Int32}
    values::Vector{Any}
end
```

where we ensure that both vectors always have the same length. In the canonical representation (the one handled by codegen and the interpreter), the edge values indicate come-from statement numbers (i.e. if edge has an entry of 15, there must be a `goto`, `gotoifnot` or implicit fall through from statement 15 that targets this phi node). Values are either SSA values or constants. It is also possible for a value to be unassigned if the variable was not defined on this path. However, undefinedness checks get explicitly inserted and represented as booleans after middle end optimizations, so code generators may assume that any use of a Phi node will have an assigned value in the corresponding slot. It is also legal for the mapping to be incomplete, i.e. for a Phi node to have missing incoming edges. In that case, it must be dynamically guaranteed that the corresponding value will not be used.

Note that SSA uses semantically occur after the terminator of the corresponding predecessor ("on the edge"). Consequently, if multiple Phi nodes appear at the start of a basic block, they are run simultaneously. This means that in the following IR snippet, if we came from block 23, %46 will take the value associated to %45 *before* we entered this block.

```
%45 = φ (#18 => %23, #23 => %50)
%46 = φ (#18 => 1.0, #23 => %45)
```

PiNodes encode statically proven information that may be implicitly assumed in basic blocks dominated by a given pi node. They are conceptually equivalent to the technique introduced in the paper ABCD: Eliminating Array Bounds Checks on Demand or the predicate info nodes in LLVM. To see how they work, consider, e.g.

```
%x::Union{Int, Float64} # %x is some Union{Int, Float64} typed ssa value
if isa(x, Int)
    # use x
else
    # use x
end
```

We can perform predicate insertion and turn this into:

```
%x::Union{Int, Float64} # %x is some Union{Int, Float64} typed ssa value
if isa(x, Int)
```

```
    %x_int = PiNode(x, Int)
    # use %x_int
else
    %x_float = PiNode(x, Float64)
    # use %x_float
end
```

Pi nodes are generally ignored in the interpreter, since they don't have any effect on the values, but they may sometimes lead to code generation in the compiler (e.g. to change from an implicitly union split representation to a plain unboxed representation). The main usefulness of PiNodes stems from the fact that path conditions of the values can be accumulated simply by def-use chain walking that is generally done for most optimizations that care about these conditions anyway.

PhiC nodes and Upsilon nodes

Exception handling complicates the SSA story moderately, because exception handling introduces additional control flow edges into the IR across which values must be tracked. One approach to do so, which is followed by LLVM, is to make calls which may throw exceptions into basic block terminators and add an explicit control flow edge to the catch handler:

```
invoke @function_that_may_throw() to label %regular unwind to %catch

regular:
# Control flow continues here

catch:
# Exceptions go here
```

However, this is problematic in a language like Julia, where at the start of the optimization pipeline, we do not know which calls throw. We would have to conservatively assume that every call (which in Julia is every statement) throws. This would have several negative effects. On the one hand, it would essentially reduce the scope of every basic block to a single call, defeating the purpose of having operations be performed at the basic block level. On the other hand, every catch basic block would have `n*m` phi node arguments (`n`, the number of statements in the critical region, `m` the number of live values through the catch block).

To work around this, we use a combination of `Upsilon` and `PhiC` nodes (the C standing for `catch`, written φ^c in the IR pretty printer, because unicode subscript c is not available). There are several ways to think of these nodes, but perhaps the easiest is to think of each `PhiC` as a load from a unique store-many, read-once slot, with `Upsilon` being the corresponding store operation. The `PhiC` has an operand list of all the upsilon nodes that store to its implicit slot. The `Upsilon` nodes however, do not record which `PhiC` node they store to. This is done for more natural integration with the rest of the SSA IR. E.g. if there are no more uses of a `PhiC` node, it is safe to delete it, and the same is true of an `Upsilon` node. In most IR passes, `PhiC` nodes can be treated like `Phi` nodes. One can follow use-def chains through them, and they can be lifted to new `PhiC` nodes and new `Upsilon` nodes (in the same places as the original `Upsilon` nodes). The result of this scheme is that the number of `Upsilon` nodes (and `PhiC` arguments) is proportional to the number of assigned values to a particular variable (before SSA conversion), rather than the number of statements in the critical region.

To see this scheme in action, consider the function

```
@noinline opaque() = invokelatest(identity, nothing) # Something opaque
function foo()
    local y
    x = 1
    try
        y = 2
        opaque()
        y = 3
        error()
    catch
    end
    (x, y)
end
```

The corresponding IR (with irrelevant types stripped) is:

```
1 ─       nothing::Nothing
2 ─ %2  = $(Expr(:enter, #4))
3 ─ %3  = ϒ (false)
│   %4  = ϒ (#undef)
│   %5  = ϒ (1)
│   %6  = ϒ (true)
│   %7  = ϒ (2)
│         invoke Main.opaque()::Any
│   %9  = ϒ (true)
│   %10 = ϒ (3)
│         invoke Main.error()::Union{}
└──       $(Expr(:unreachable))::Union{}
4 ┄ %13 = φᶜ (%3, %6, %9)::Bool
│   %14 = φᶜ (%4, %7, %10)::Core.Compiler.MaybeUndef(Int64)
│   %15 = φᶜ (%5)::Core.Const(1)
└──       $(Expr(:leave, Core.SSAValue(2)))
5 ─       $(Expr(:pop_exception, :(%2)))::Any
│         $(Expr(:throw_undef_if_not, :y, :(%13)))::Any
│   %19 = Core.tuple(%15, %14)
└──       return %19
```

Note in particular that every value live into the critical region gets an upsilon node at the top of the critical region. This is because catch blocks are considered to have an invisible control flow edge from outside the function. As a result, no SSA value dominates the catch blocks, and all incoming values have to come through a `φᶜ` node.

Main SSA data structure

The main SSAIR data structure is worthy of discussion. It draws inspiration from LLVM and Webkit's B3 IR. The core of the data structure is a flat vector of statements. Each statement is implicitly assigned an SSA value based on its position in the vector (i.e. the result of the statement at idx 1 can be accessed using `SSAValue(1)` etc). For each SSA value, we additionally maintain its type. Since, SSA values are definitionally assigned only once, this type is also the result type of the expression at the corresponding index. However, while this representation is rather efficient (since the assignments don't need to be explicitly encoded), it of course carries the drawback that order is semantically significant, so reorderings

and insertions change statement numbers. Additionally, we do not keep use lists (i.e. it is impossible to walk from a def to all its uses without explicitly computing this map–def lists however are trivial since you can look up the corresponding statement from the index), so the LLVM-style RAUW (replace-all-uses-with) operation is unavailable.

Instead, we do the following:

- We keep a separate buffer of nodes to insert (including the position to insert them at, the type of the corresponding value and the node itself). These nodes are numbered by their occurrence in the insertion buffer, allowing their values to be immediately used elsewhere in the IR (i.e. if there are 12 statements in the original statement list, the first new statement will be accessible as `SSAValue(13)`).
- RAUW style operations are performed by setting the corresponding statement index to the replacement value.
- Statements are erased by setting the corresponding statement to `nothing` (this is essentially just a special-case convention of the above).
- If there are any uses of the statement being erased, they will be set to `nothing`.

There is a `compact!` function that compacts the above data structure by performing the insertion of nodes in the appropriate place, trivial copy propagation, and renaming of uses to any changed SSA values. However, the clever part of this scheme is that this compaction can be done lazily as part of the subsequent pass. Most optimization passes need to walk over the entire list of statements, performing analysis or modifications along the way. We provide an `IncrementalCompact` iterator that can be used to iterate over the statement list. It will perform any necessary compaction and return the new index of the node, as well as the node itself. It is legal at this point to walk def-use chains, as well as make any modifications or deletions to the IR (insertions are disallowed however).

The idea behind this arrangement is that, since the optimization passes need to touch the corresponding memory anyway and incur the corresponding memory access penalty, performing the extra housekeeping should have comparatively little overhead (and save the overhead of maintaining these data structures during IR modification).

108.25 EscapeAnalysis

`Compiler.EscapeAnalysis` is a compiler utility module that aims to analyze escape information of Julia's SSA-form IR a.k.a. `IRCode`.

This escape analysis aims to:

- leverage Julia's high-level semantics, especially reason about escapes and aliasing via inter-procedural calls
- be versatile enough to be used for various optimizations including alias-aware SROA, early `finalize` insertion, copy-free `ImmutableArray` construction, stack allocation of mutable objects, and so on.
- achieve a simple implementation based on a fully backward data-flow analysis implementation as well as a new lattice design that combines orthogonal lattice properties

Try it out!

You can give a try to the escape analysis by loading the `EAUtils.jl` utility script that defines the convenience entries `code_escapes` and `@code_escapes` for testing and debugging purposes:

```
julia> # InteractiveUtils.@activate Compiler # to use the stdlib version of the Compiler

       let JULIA_DIR = normpath(Sys.BINDIR, Base.DATAROOTDIR, "julia")
           include(normpath(JULIA_DIR, "Compiler", "test", "EAUtils.jl"))
           using .EAUtils
       end

julia> mutable struct SafeRef{T}
           x::T
       end

julia> Base.getindex(x::SafeRef) = x.x;

julia> Base.setindex!(x::SafeRef, v) = x.x = v;

julia> Base.isassigned(x::SafeRef) = true;

julia> get′(x) = isassigned(x) ? x[] : throw(x);

julia> result = code_escapes((Base.RefValue{String},String,String,)) do r1, s2, s3
           r2 = Ref(s2)
           r3 = SafeRef(s3)
           try
               s1 = get′(r1)
               ret = sizeof(s1)
           catch err
               global GV = err # `r1` may escape
           end
           s2 = get′(r2)       # `r2` doesn't escape
           s3 = get′(r3)       # `r3` doesn't escape
           return s2, s3, s4
       end
#2(X r1::Base.RefValue{String}, ✓ s2::String, ✓ s3::String) in Main at REPL[7]:2
✓′ 1 ── %1  = %new(Base.RefValue{String}, _3)::Base.RefValue{String}
✓′ └─── %2  = %new(Main.SafeRef{String}, _4)::Main.SafeRef{String}
✓′ 2 ── %3  = ϒ (%2)::Main.SafeRef{String}
✓′ │    %4  = ϒ (%1)::Core.PartialStruct(Base.RefValue{String}, Union{Nothing, Bool}[0],
↪  Any[String])
◌  └─── %5  = enter #8
◌  3 ── %6  =   builtin Base.isdefined(_2, :x)::Bool
◌  └─       goto #5 if not %6
◌  4 ──       goto #6
◌  5 ──         builtin Main.throw(_2)::Union{}
◌  └─       unreachable
◌  6 ──       $(Expr(:leave, :(%5)))
◌  7 ──       goto #11
✓′ 8 ┄─ %13 = φᶜ (%3)::Main.SafeRef{String}
✓′ │    %14 = φᶜ (%4)::Core.PartialStruct(Base.RefValue{String}, Union{Nothing, Bool}[0],
↪  Any[String])
```

```
X  └──── %15 = $(Expr(:the_exception))::Base.RefValue{String}
◌  9 ──       nothing::Nothing
X  10 ─         builtin Core.setglobal!(Main, :GV, %15)::Base.RefValue{String}
◌  └────      $(Expr(:pop_exception, :(%5)))::Core.Const(nothing)
✓′ 11 ┄ %19 = φ (#7 => %2, #10 => %13)::Main.SafeRef{String}
✓′ │    %20 = φ (#7 => %1, #10 => %14)::Core.PartialStruct(Base.RefValue{String}, Union{Nothing,
↪  Bool}[0], Any[String])
✓  │    %21 =   builtin Base.getfield(%20, :x)::String
✓  │    %22 =   builtin Base.getfield(%19, :x)::String
X  │    %23 = Main.s4::Any
↑′ │    %24 =   builtin Core.tuple(%21, %22, %23)::Tuple{String, String, Any}
◌  └────      return %24
```

The symbols on the side of each call argument and SSA statements represent the following meaning:

- ◌ (plain): this value is not analyzed because escape information of it won't be used anyway (when the object is `isbitstype` for example)
- ✓ (green or cyan): this value never escapes (`has_no_escape(result.state[x])` holds), colored blue if it has arg escape also (`has_arg_escape(result.state[x])` holds)
- ↑ (blue or yellow): this value can escape to the caller via return (`has_return_escape(result.state[x])` holds), colored yellow if it has unhandled thrown escape also (`has_thrown_escape(result.state[x])` holds)
- X (red): this value can escape to somewhere the escape analysis can't reason about like escapes to a global memory (`has_all_escape(result.state[x])` holds)
- * (bold): this value's escape state is between the `ReturnEscape` and `AllEscape` in the partial order of `EscapeInfo`, colored yellow if it has unhandled thrown escape also (`has_thrown_escape(result.state[x])` holds)
- ′: this value has additional object field / array element information in its `AliasInfo` property

Escape information of each call argument and SSA value can be inspected programmatically as like:

```
julia> result.state[Core.Argument(3)] # get EscapeInfo of `s2`
ArgEscape

julia> result.state[Core.SSAValue(3)] # get EscapeInfo of `r3`
NoEscape′
```

Analysis Design

Lattice Design

`EscapeAnalysis` is implemented as a data-flow analysis that works on a lattice of `x::EscapeInfo`, which is composed of the following properties:

- `x.Analyzed::Bool`: not formally part of the lattice, only indicates x has not been analyzed or not
- `x.ReturnEscape::BitSet`: records SSA statements where x can escape to the caller via return

- `x.ThrownEscape::BitSet`: records SSA statements where `x` can be thrown as exception (used for the exception handling described below)
- `x.AliasInfo`: maintains all possible values that can be aliased to fields or array elements of `x` (used for the alias analysis described below)
- `x.ArgEscape::Int` (not implemented yet): indicates it will escape to the caller through `setfield!` on argument(s)

These attributes can be combined to create a partial lattice that has a finite height, given the invariant that an input program has a finite number of statements, which is assured by Julia's semantics. The clever part of this lattice design is that it enables a simpler implementation of lattice operations by allowing them to handle each lattice property separately[1].

Backward Escape Propagation

This escape analysis implementation is based on the data-flow algorithm described in the paper[2]. The analysis works on the lattice of `EscapeInfo` and transitions lattice elements from the bottom to the top until every lattice element gets converged to a fixed point by maintaining a (conceptual) working set that contains program counters corresponding to remaining SSA statements to be analyzed. The analysis manages a single global state that tracks `EscapeInfo` of each argument and SSA statement, but also note that some flow-sensitivity is encoded as program counters recorded in `EscapeInfo`'s `ReturnEscape` property, which can be combined with domination analysis later to reason about flow-sensitivity if necessary.

One distinctive design of this escape analysis is that it is fully *backward*, i.e. escape information flows *from usages to definitions*. For example, in the code snippet below, EA first analyzes the statement `return %1` and imposes `ReturnEscape` on `%1` (corresponding to `obj`), and then it analyzes `%1 = %new(Base.RefValue{Base.RefValue{Strin` `_2}))` and propagates the `ReturnEscape` imposed on `%1` to the call argument `_2` (corresponding to `s`):

```
julia> code_escapes((Base.RefValue{String},)) do s
           obj = Ref(s)
           return obj
       end
#5(↑ s::Base.RefValue{String}) in Main at REPL[1]:2
↑′ 1 ─ %1 = %new(Base.RefValue{Base.RefValue{String}}, _2)::Base.RefValue{Base.RefValue{String}}
◌  └──      return %1
```

The key observation here is that this backward analysis allows escape information to flow naturally along the use-def chain rather than control-flow[3]. As a result this scheme enables a simple implementation of escape analysis, e.g. `PhiNode` for example can be handled simply by propagating escape information imposed on a `PhiNode` to its predecessor values:

```
julia> code_escapes((Bool, Base.RefValue{String}, Base.RefValue{String})) do cnd, s, t
           if cnd
               obj = Ref(s)
           else
               obj = Ref(t)
           end
           return obj
       end
#8(✓ cnd::Bool, ↑ s::Base.RefValue{String}, ↑ t::Base.RefValue{String}) in Main at REPL[1]:2
```

```
◌  1 ─      goto #3 if not _2
↑′ 2 ─ %2 = %new(Base.RefValue{Base.RefValue{String}}, _3)::Base.RefValue{Base.RefValue{String}}
◌  └──      goto #4
↑′ 3 ─ %4 = %new(Base.RefValue{Base.RefValue{String}}, _4)::Base.RefValue{Base.RefValue{String}}
↑′ 4 ┄ %5 = φ (#2 => %2, #3 => %4)::Core.PartialStruct(Base.RefValue{Base.RefValue{String}},
↪  Union{Nothing, Bool}[0], Any[Base.RefValue{String}])
◌  └──      return %5
```

Alias Analysis

`EscapeAnalysis` implements a backward field analysis in order to reason about escapes imposed on object fields with certain accuracy, and `x::EscapeInfo`'s `x.AliasInfo` property exists for this purpose. It records all possible values that can be aliased to fields of `x` at "usage" sites, and then the escape information of that recorded values are propagated to the actual field values later at "definition" sites. More specifically, the analysis records a value that may be aliased to a field of object by analyzing `getfield` call, and then it propagates its escape information to the field when analyzing `%new(...)` expression or `setfield!` call[4].

```
julia> code_escapes((String,)) do s
           obj = SafeRef("init")
           obj[] = s
           v = obj[]
           return v
       end
#11(✓ s::String) in Main at REPL[1]:2
◌  1 ─     return _2
```

In the example above, `ReturnEscape` imposed on `%3` (corresponding to `v`) is *not* directly propagated to `%1` (corresponding to `obj`) but rather that `ReturnEscape` is only propagated to `_2` (corresponding to `s`). Here `%3` is recorded in `%1`'s `AliasInfo` property as it can be aliased to the first field of `%1`, and then when analyzing `Base.setfield!(%1, :x, _2)::String`, that escape information is propagated to `_2` but not to `%1`.

So `EscapeAnalysis` tracks which IR elements can be aliased across a `getfield`-`%new`/`setfield!` chain in order to analyze escapes of object fields, but actually this alias analysis needs to be generalized to handle other IR elements as well. This is because in Julia IR the same object is sometimes represented by different IR elements and so we should make sure that those different IR elements that actually can represent the same object share the same escape information. IR elements that return the same object as their operand(s), such as `PiNode` and `typeassert`, can cause that IR-level aliasing and thus requires escape information imposed on any of such aliased values to be shared between them. More interestingly, it is also needed for correctly reasoning about mutations on `PhiNode`. Let's consider the following example:

```
julia> code_escapes((Bool, String,)) do cond, x
           if cond
               ϕ2 = ϕ1 = SafeRef("foo")
           else
               ϕ2 = ϕ1 = SafeRef("bar")
           end
           ϕ2[] = x
           y = ϕ1[]
           return y
       end
#14(✓ cond::Bool, ✓ x::String) in Main at REPL[1]:2
```

```
◌  1 ─      goto #3 if not _2
✓′ 2 ─ %2 = %new(Main.SafeRef{String}, "foo")::Main.SafeRef{String}
◌  └──      goto #4
✓′ 3 ─ %4 = %new(Main.SafeRef{String}, "bar")::Main.SafeRef{String}
✓′ 4 ┄ %5 = φ (#2 => %2, #3 => %4)::Main.SafeRef{String}
✓′ │   %6 = φ (#2 => %2, #3 => %4)::Main.SafeRef{String}
◌  │          builtin Base.setfield!(%5, :x, _3)::String
✓  │   %8 =   builtin Base.getfield(%6, :x)::String
◌  └──      return %8
```

`ϕ1 = %5` and `ϕ2 = %6` are aliased and thus `ReturnEscape` imposed on `%8 = Base.getfield(%6, :x)::String` (corresponding to `y = ϕ1[]`) needs to be propagated to `Base.setfield!(%5, :x, _3)::String` (corresponding to `ϕ2[] = x`). In order for such escape information to be propagated correctly, the analysis should recognize that the *predecessors* of `ϕ1` and `ϕ2` can be aliased as well and equalize their escape information.

One interesting property of such aliasing information is that it is not known at "usage" site but can only be derived at "definition" site (as aliasing is conceptually equivalent to assignment), and thus it doesn't naturally fit in a backward analysis. In order to efficiently propagate escape information between related values, EscapeAnalysis.jl uses an approach inspired by the escape analysis algorithm explained in an old JVM paper[5]. That is, in addition to managing escape lattice elements, the analysis also maintains an "equi"-alias set, a disjoint set of aliased arguments and SSA statements. The alias set manages values that can be aliased to each other and allows escape information imposed on any of such aliased values to be equalized between them.

Array Analysis

The alias analysis for object fields described above can also be generalized to analyze array operations. `EscapeAnalysis` implements handlings for various primitive array operations so that it can propagate escapes via `arrayref`-`arrayset` use-def chain and does not escape allocated arrays too conservatively:

```
julia> code_escapes((String,)) do s
           ary = Any[]
           push!(ary, SafeRef(s))
           return ary[1], length(ary)
       end
#17(✓ s::String) in Main at REPL[1]:2
X  1 ── %1  =   builtin Core.memorynew(Memory{Any}, 0)::Memory{Any}
X  │    %2  =   builtin Core.memoryrefnew(%1)::MemoryRef{Any}
X  │    %3  = %new(Vector{Any}, %2, (0,))::Vector{Any}
X  │    %4  = %new(Main.SafeRef{String}, _2)::Main.SafeRef{String}
X  │    %5  =   builtin Base.getfield(%3, :ref)::MemoryRef{Any}
X  │    %6  =   builtin Base.getfield(%5, :mem)::Memory{Any}
X  │    %7  =   builtin Base.getfield(%6, :length)::Int64
X  │    %8  =   builtin Base.getfield(%3, :size)::Tuple{Int64}
X  │    %9  =   builtin Core.getfield(%8, 1)::Int64
◌  │    %10 = intrinsic Base.add_int(%9, 1)::Int64
◌  │    %11 =   builtin Base.memoryrefoffset(%5)::Int64
◌  │    %12 = intrinsic Base.add_int(%11, %10)::Int64
◌  │    %13 = intrinsic Base.sub_int(%12, 1)::Int64
◌  │    %14 = intrinsic Base.slt_int(%7, %13)::Bool
◌  └────       goto #3 if not %14
◌  2 ──          invoke Base._growend_internal!(%3::Vector{Any}, 1::Int64, %9::Int64)::Nothing
```

```
X  3 ─── %17 =   builtin Core.tuple(%10)::Tuple{Int64}
◌  │            builtin Base.setfield!(%3, :size, %17)::Tuple{Int64}
◌  └────       goto #4
X  4 ── %20 =   builtin Base.getfield(%3, :size)::Tuple{Int64}
X  │    %21 =   builtin Core.getfield(%20, 1)::Int64
X  │    %22 =   builtin Base.getfield(%3, :ref)::MemoryRef{Any}
X  │    %23 =   builtin Base.memoryrefnew(%22, %21, false)::MemoryRef{Any}
X  │            builtin Base.memoryrefset!(%23, %4, :not_atomic, false)::Main.SafeRef{String}
◌  └────       goto #5
◌  5 ── %26 = $(Expr(:boundscheck, true))::Bool
◌  └────       goto #10 if not %26
◌  6 ── %28 = intrinsic Base.sub_int(1, 1)::Int64
◌  │    %29 = intrinsic Base.bitcast(Base.UInt, %28)::UInt64
X  │    %30 =   builtin Base.getfield(%3, :size)::Tuple{Int64}
X  │    %31 =   builtin Core.getfield(%30, 1)::Int64
◌  │    %32 = intrinsic Base.bitcast(Base.UInt, %31)::UInt64
◌  │    %33 = intrinsic Base.ult_int(%29, %32)::Bool
◌  └────       goto #8 if not %33
◌  7 ──        goto #9
◌  8 ── %36 =   builtin Core.tuple(1)::Tuple{Int64}
✓  │             invoke Base.throw_boundserror(%3::Vector{Any}, %36::Tuple{Int64})::Union{}
◌  └────       unreachable
◌  9 ──        nothing::Nothing
X  10 ─ %40 =   builtin Base.getfield(%3, :ref)::MemoryRef{Any}
X  │    %41 =   builtin Base.memoryrefnew(%40, 1, false)::MemoryRef{Any}
X  │    %42 =   builtin Base.memoryrefget(%41, :not_atomic, false)::Any
◌  └────       goto #11
X  11 ─ %44 =   builtin Base.getfield(%3, :size)::Tuple{Int64}
X  │    %45 =   builtin Core.getfield(%44, 1)::Int64
↑′ │    %46 =   builtin Core.tuple(%42, %45)::Tuple{Any, Int64}
◌  └────       return %46
```

In the above example `EscapeAnalysis` understands that `%20` and `%2` (corresponding to the allocated object `SafeRef(s)`) are aliased via the `arrayset-arrayref` chain and imposes `ReturnEscape` on them, but not impose it on the allocated array `%1` (corresponding to `ary`). `EscapeAnalysis` still imposes `ThrownEscape` on `ary` since it also needs to account for potential escapes via `BoundsError`, but also note that such unhandled `ThrownEscape` can often be ignored when optimizing the `ary` allocation.

Furthermore, in cases when array index information as well as array dimensions can be known *precisely*, `EscapeAnalysis` is able to even reason about "per-element" aliasing via `arrayref-arrayset` chain, as `EscapeAnalysis` does "per-field" alias analysis for objects:

```
julia> code_escapes((String,String)) do s, t
           ary = Vector{Any}(undef, 2)
           ary[1] = SafeRef(s)
           ary[2] = SafeRef(t)
           return ary[1], length(ary)
       end
#20(✓ s::String, ✓ t::String) in Main at REPL[1]:2
X  1 ── %1  =   builtin Core.memorynew(Memory{Any}, 2)::Core.PartialStruct(Memory{Any},
↪  Union{Nothing, Bool}[0, 0], Any[Core.Const(2), Ptr{Nothing}])
X  │    %2  =   builtin Core.memoryrefnew(%1)::MemoryRef{Any}
```

```
X  │    %3  = %new(Vector{Any}, %2, (2,))::Vector{Any}
X  │    %4  = %new(Main.SafeRef{String}, _2)::Main.SafeRef{String}
◌  │    %5  = $(Expr(:boundscheck, true))::Bool
◌  └────       goto #6 if not %5
◌  2 ── %7  = intrinsic Base.sub_int(1, 1)::Int64
◌  │    %8  = intrinsic Base.bitcast(Base.UInt, %7)::UInt64
X  │    %9  =   builtin Base.getfield(%3, :size)::Tuple{Int64}
X  │    %10 =   builtin Core.getfield(%9, 1)::Int64
◌  │    %11 = intrinsic Base.bitcast(Base.UInt, %10)::UInt64
◌  │    %12 = intrinsic Base.ult_int(%8, %11)::Bool
◌  └────       goto #4 if not %12
◌  3 ──       goto #5
◌  4 ── %15 =   builtin Core.tuple(1)::Tuple{Int64}
✓  │             invoke Base.throw_boundserror(%3::Vector{Any}, %15::Tuple{Int64})::Union{}
◌  └────       unreachable
◌  5 ──       nothing::Nothing
X  6 ┄─ %19 =   builtin Base.getfield(%3, :ref)::MemoryRef{Any}
X  │    %20 =   builtin Base.memoryrefnew(%19, 1, false)::MemoryRef{Any}
X  │             builtin Base.memoryrefset!(%20, %4, :not_atomic, false)::Main.SafeRef{String}
◌  └────       goto #7
X  7 ── %23 = %new(Main.SafeRef{String}, _3)::Main.SafeRef{String}
◌  │    %24 = $(Expr(:boundscheck, true))::Bool
◌  └────       goto #12 if not %24
◌  8 ── %26 = intrinsic Base.sub_int(2, 1)::Int64
◌  │    %27 = intrinsic Base.bitcast(Base.UInt, %26)::UInt64
X  │    %28 =   builtin Base.getfield(%3, :size)::Tuple{Int64}
X  │    %29 =   builtin Core.getfield(%28, 1)::Int64
◌  │    %30 = intrinsic Base.bitcast(Base.UInt, %29)::UInt64
◌  │    %31 = intrinsic Base.ult_int(%27, %30)::Bool
◌  └────       goto #10 if not %31
◌  9 ──       goto #11
◌  10 ─ %34 =   builtin Core.tuple(2)::Tuple{Int64}
✓  │             invoke Base.throw_boundserror(%3::Vector{Any}, %34::Tuple{Int64})::Union{}
◌  └────       unreachable
◌  11 ─       nothing::Nothing
X  12 ┄ %38 =   builtin Base.getfield(%3, :ref)::MemoryRef{Any}
X  │    %39 =   builtin Base.memoryrefnew(%38, 2, false)::MemoryRef{Any}
X  │             builtin Base.memoryrefset!(%39, %23, :not_atomic, false)::Main.SafeRef{String}
◌  └────       goto #13
◌  13 ─ %42 = $(Expr(:boundscheck, true))::Bool
◌  └────       goto #18 if not %42
◌  14 ─ %44 = intrinsic Base.sub_int(1, 1)::Int64
◌  │    %45 = intrinsic Base.bitcast(Base.UInt, %44)::UInt64
X  │    %46 =   builtin Base.getfield(%3, :size)::Tuple{Int64}
X  │    %47 =   builtin Core.getfield(%46, 1)::Int64
◌  │    %48 = intrinsic Base.bitcast(Base.UInt, %47)::UInt64
◌  │    %49 = intrinsic Base.ult_int(%45, %48)::Bool
◌  └────       goto #16 if not %49
◌  15 ─       goto #17
◌  16 ─ %52 =   builtin Core.tuple(1)::Tuple{Int64}
✓  │             invoke Base.throw_boundserror(%3::Vector{Any}, %52::Tuple{Int64})::Union{}
◌  └────       unreachable
◌  17 ─       nothing::Nothing
```

```
X  18 ┄ %56 =   builtin Base.getfield(%3, :ref)::MemoryRef{Any}
X  │    %57 =   builtin Base.memoryrefnew(%56, 1, false)::MemoryRef{Any}
X  │    %58 =   builtin Base.memoryrefget(%57, :not_atomic, false)::Any
◌  └────       goto #19
X  19 ─ %60 =   builtin Base.getfield(%3, :size)::Tuple{Int64}
X  │    %61 =   builtin Core.getfield(%60, 1)::Int64
↑′ │    %62 =   builtin Core.tuple(%58, %61)::Tuple{Any, Int64}
◌  └────       return %62
```

Note that `ReturnEscape` is only imposed on `%2` (corresponding to `SafeRef(s)`) but not on `%4` (corresponding to `SafeRef(t)`). This is because the allocated array's dimension and indices involved with all `arrayref`/`arrayset` operations are available as constant information and `EscapeAnalysis` can understand that `%6` is aliased to `%2` but never be aliased to `%4`. In this kind of case, the succeeding optimization passes will be able to replace `Base.arrayref(true, %1, 1)::Any` with `%2` (a.k.a. "load-forwarding") and eventually eliminate the allocation of array `%1` entirely (a.k.a. "scalar-replacement").

When compared to object field analysis, where an access to object field can be analyzed trivially using type information derived by inference, array dimension isn't encoded as type information and so we need an additional analysis to derive that information. `EscapeAnalysis` at this moment first does an additional simple linear scan to analyze dimensions of allocated arrays before firing up the main analysis routine so that the succeeding escape analysis can precisely analyze operations on those arrays.

However, such precise "per-element" alias analysis is often hard. Essentially, the main difficulty inherit to array is that array dimension and index are often non-constant:

- loop often produces loop-variant, non-constant array indices
- (specific to vectors) array resizing changes array dimension and invalidates its constant-ness

Let's discuss those difficulties with concrete examples.

In the following example, `EscapeAnalysis` fails the precise alias analysis since the index at the `Base.arrayset(false, %4, %8, %6)::Vector{Any}` is not (trivially) constant. Especially `Any[nothing, nothing]` forms a loop and calls that `arrayset` operation in a loop, where `%6` is represented as a ϕ-node value (whose value is control-flow dependent). As a result, `ReturnEscape` ends up imposed on both `%23` (corresponding to `SafeRef(s)`) and `%25` (corresponding to `SafeRef(t)`), although ideally we want it to be imposed only on `%23` but not on `%25`:

```
julia> code_escapes((String,String)) do s, t
           ary = Any[nothing, nothing]
           ary[1] = SafeRef(s)
           ary[2] = SafeRef(t)
           return ary[1], length(ary)
       end
#23(✓ s::String, ✓ t::String) in Main at REPL[1]:2
X  1 ── %1  =   builtin Core.memorynew(Memory{Any}, 2)::Memory{Any}
X  │    %2  =   builtin Core.memoryrefnew(%1)::MemoryRef{Any}
X  └──── %3  = %new(Vector{Any}, %2, (2,))::Vector{Any}
◌  2 ┄── %4  = φ (#1 => 1, #6 => %14)::Int64
◌  │    %5  = φ (#1 => 1, #6 => %15)::Int64
X  │    %6  =   builtin Base.getfield(%3, :ref)::MemoryRef{Any}
```

```
X  │    %7  =   builtin Base.memoryrefnew(%6, %4, false)::MemoryRef{Any}
◌  │            builtin Base.memoryrefset!(%7, nothing, :not_atomic, false)::Nothing
◌  │    %9  =   builtin (%5 === 2)::Bool
◌  └──        goto #4 if not %9
◌  3 ──        goto #5
◌  4 ── %12 = intrinsic Base.add_int(%5, 1)::Int64
◌  └──        goto #5
◌  5 ┄─ %14 = φ (#4 => %12)::Int64
◌  │    %15 = φ (#4 => %12)::Int64
◌  │    %16 = φ (#3 => true, #4 => false)::Bool
◌  │    %17 = intrinsic Base.not_int(%16)::Bool
◌  └──        goto #7 if not %17
◌  6 ──        goto #2
◌  7 ──        goto #8
X  8 ── %21 = %new(Main.SafeRef{String}, _2)::Main.SafeRef{String}
◌  │    %22 = $(Expr(:boundscheck, true))::Bool
◌  └──        goto #13 if not %22
◌  9 ── %24 = intrinsic Base.sub_int(1, 1)::Int64
◌  │    %25 = intrinsic Base.bitcast(Base.UInt, %24)::UInt64
X  │    %26 =   builtin Base.getfield(%3, :size)::Tuple{Int64}
X  │    %27 =   builtin Core.getfield(%26, 1)::Int64
◌  │    %28 = intrinsic Base.bitcast(Base.UInt, %27)::UInt64
◌  │    %29 = intrinsic Base.ult_int(%25, %28)::Bool
◌  └──        goto #11 if not %29
◌  10 ─        goto #12
◌  11 ─ %32 =   builtin Core.tuple(1)::Tuple{Int64}
✓  │             invoke Base.throw_boundserror(%3::Vector{Any}, %32::Tuple{Int64})::Union{}
◌  └──        unreachable
◌  12 ─        nothing::Nothing
X  13 ┄ %36 =   builtin Base.getfield(%3, :ref)::MemoryRef{Any}
X  │    %37 =   builtin Base.memoryrefnew(%36, 1, false)::MemoryRef{Any}
X  │            builtin Base.memoryrefset!(%37, %21, :not_atomic, false)::Main.SafeRef{String}
◌  └──        goto #14
X  14 ─ %40 = %new(Main.SafeRef{String}, _3)::Main.SafeRef{String}
◌  │    %41 = $(Expr(:boundscheck, true))::Bool
◌  └──        goto #19 if not %41
◌  15 ─ %43 = intrinsic Base.sub_int(2, 1)::Int64
◌  │    %44 = intrinsic Base.bitcast(Base.UInt, %43)::UInt64
X  │    %45 =   builtin Base.getfield(%3, :size)::Tuple{Int64}
X  │    %46 =   builtin Core.getfield(%45, 1)::Int64
◌  │    %47 = intrinsic Base.bitcast(Base.UInt, %46)::UInt64
◌  │    %48 = intrinsic Base.ult_int(%44, %47)::Bool
◌  └──        goto #17 if not %48
◌  16 ─        goto #18
◌  17 ─ %51 =   builtin Core.tuple(2)::Tuple{Int64}
✓  │             invoke Base.throw_boundserror(%3::Vector{Any}, %51::Tuple{Int64})::Union{}
◌  └──        unreachable
◌  18 ─        nothing::Nothing
X  19 ┄ %55 =   builtin Base.getfield(%3, :ref)::MemoryRef{Any}
X  │    %56 =   builtin Base.memoryrefnew(%55, 2, false)::MemoryRef{Any}
X  │            builtin Base.memoryrefset!(%56, %40, :not_atomic, false)::Main.SafeRef{String}
◌  └──        goto #20
◌  20 ─ %59 = $(Expr(:boundscheck, true))::Bool
```

```
◌  └──       goto #25 if not %59
◌  21 ─ %61 = intrinsic Base.sub_int(1, 1)::Int64
◌  │    %62 = intrinsic Base.bitcast(Base.UInt, %61)::UInt64
X  │    %63 =   builtin Base.getfield(%3, :size)::Tuple{Int64}
X  │    %64 =   builtin Core.getfield(%63, 1)::Int64
◌  │    %65 = intrinsic Base.bitcast(Base.UInt, %64)::UInt64
◌  │    %66 = intrinsic Base.ult_int(%62, %65)::Bool
◌  └──       goto #23 if not %66
◌  22 ─       goto #24
◌  23 ─ %69 =   builtin Core.tuple(1)::Tuple{Int64}
✓  │             invoke Base.throw_boundserror(%3::Vector{Any}, %69::Tuple{Int64})::Union{}
◌  └──       unreachable
◌  24 ─       nothing::Nothing
X  25 ┄ %73 =   builtin Base.getfield(%3, :ref)::MemoryRef{Any}
X  │    %74 =   builtin Base.memoryrefnew(%73, 1, false)::MemoryRef{Any}
X  │    %75 =   builtin Base.memoryrefget(%74, :not_atomic, false)::Any
◌  └──       goto #26
X  26 ─ %77 =   builtin Base.getfield(%3, :size)::Tuple{Int64}
X  │    %78 =   builtin Core.getfield(%77, 1)::Int64
↑′ │    %79 =   builtin Core.tuple(%75, %78)::Tuple{Any, Int64}
◌  └──       return %79
```

The next example illustrates how vector resizing makes precise alias analysis hard. The essential difficulty is that the dimension of allocated array `%1` is first initialized as `0`, but it changes by the two `:jl_array_grow_end` calls afterwards. `EscapeAnalysis` currently simply gives up precise alias analysis whenever it encounters any array resizing operations and so `ReturnEscape` is imposed on both `%2` (corresponding to `SafeRef(s)`) and `%20` (corresponding to `SafeRef(t)`):

```
julia> code_escapes((String,String)) do s, t
           ary = Any[]
           push!(ary, SafeRef(s))
           push!(ary, SafeRef(t))
           ary[1], length(ary)
       end
#26(✓ s::String, ✓ t::String) in Main at REPL[1]:2
X  1 ── %1  =   builtin Core.memorynew(Memory{Any}, 0)::Memory{Any}
X  │    %2  =   builtin Core.memoryrefnew(%1)::MemoryRef{Any}
X  │    %3  = %new(Vector{Any}, %2, (0,))::Vector{Any}
X  │    %4  = %new(Main.SafeRef{String}, _2)::Main.SafeRef{String}
X  │    %5  =   builtin Base.getfield(%3, :ref)::MemoryRef{Any}
X  │    %6  =   builtin Base.getfield(%5, :mem)::Memory{Any}
X  │    %7  =   builtin Base.getfield(%6, :length)::Int64
X  │    %8  =   builtin Base.getfield(%3, :size)::Tuple{Int64}
X  │    %9  =   builtin Core.getfield(%8, 1)::Int64
◌  │    %10 = intrinsic Base.add_int(%9, 1)::Int64
◌  │    %11 =   builtin Base.memoryrefoffset(%5)::Int64
◌  │    %12 = intrinsic Base.add_int(%11, %10)::Int64
◌  │    %13 = intrinsic Base.sub_int(%12, 1)::Int64
◌  │    %14 = intrinsic Base.slt_int(%7, %13)::Bool
◌  └──       goto #3 if not %14
◌  2 ──        invoke Base._growend_internal!(%3::Vector{Any}, 1::Int64, %9::Int64)::Nothing
X  3 ┄─ %17 =   builtin Core.tuple(%10)::Tuple{Int64}
```

```
◌  │           builtin Base.setfield!(%3, :size, %17)::Tuple{Int64}
◌  └──       goto #4
X  4 ── %20 =   builtin Base.getfield(%3, :size)::Tuple{Int64}
X  │    %21 =   builtin Core.getfield(%20, 1)::Int64
X  │    %22 =   builtin Base.getfield(%3, :ref)::MemoryRef{Any}
X  │    %23 =   builtin Base.memoryrefnew(%22, %21, false)::MemoryRef{Any}
X  │           builtin Base.memoryrefset!(%23, %4, :not_atomic, false)::Main.SafeRef{String}
◌  └──       goto #5
X  5 ── %26 = %new(Main.SafeRef{String}, _3)::Main.SafeRef{String}
X  │    %27 =   builtin Base.getfield(%3, :ref)::MemoryRef{Any}
X  │    %28 =   builtin Base.getfield(%27, :mem)::Memory{Any}
X  │    %29 =   builtin Base.getfield(%28, :length)::Int64
X  │    %30 =   builtin Base.getfield(%3, :size)::Tuple{Int64}
X  │    %31 =   builtin Core.getfield(%30, 1)::Int64
◌  │    %32 = intrinsic Base.add_int(%31, 1)::Int64
◌  │    %33 =   builtin Base.memoryrefoffset(%27)::Int64
◌  │    %34 = intrinsic Base.add_int(%33, %32)::Int64
◌  │    %35 = intrinsic Base.sub_int(%34, 1)::Int64
◌  │    %36 = intrinsic Base.slt_int(%29, %35)::Bool
◌  └──       goto #7 if not %36
◌  6 ──        invoke Base._growend_internal!(%3::Vector{Any}, 1::Int64, %31::Int64)::Nothing
X  7 ┄─ %39 =   builtin Core.tuple(%32)::Tuple{Int64}
◌  │           builtin Base.setfield!(%3, :size, %39)::Tuple{Int64}
◌  └──       goto #8
X  8 ── %42 =   builtin Base.getfield(%3, :size)::Tuple{Int64}
X  │    %43 =   builtin Core.getfield(%42, 1)::Int64
X  │    %44 =   builtin Base.getfield(%3, :ref)::MemoryRef{Any}
X  │    %45 =   builtin Base.memoryrefnew(%44, %43, false)::MemoryRef{Any}
X  │           builtin Base.memoryrefset!(%45, %26, :not_atomic, false)::Main.SafeRef{String}
◌  └──       goto #9
◌  9 ── %48 = $(Expr(:boundscheck, true))::Bool
◌  └──       goto #14 if not %48
◌  10 ─ %50 = intrinsic Base.sub_int(1, 1)::Int64
◌  │    %51 = intrinsic Base.bitcast(Base.UInt, %50)::UInt64
X  │    %52 =   builtin Base.getfield(%3, :size)::Tuple{Int64}
X  │    %53 =   builtin Core.getfield(%52, 1)::Int64
◌  │    %54 = intrinsic Base.bitcast(Base.UInt, %53)::UInt64
◌  │    %55 = intrinsic Base.ult_int(%51, %54)::Bool
◌  └──       goto #12 if not %55
◌  11 ─      goto #13
◌  12 ─ %58 =   builtin Core.tuple(1)::Tuple{Int64}
✓  │           invoke Base.throw_boundserror(%3::Vector{Any}, %58::Tuple{Int64})::Union{}
◌  └──       unreachable
◌  13 ─      nothing::Nothing
X  14 ┄ %62 =   builtin Base.getfield(%3, :ref)::MemoryRef{Any}
X  │    %63 =   builtin Base.memoryrefnew(%62, 1, false)::MemoryRef{Any}
X  │    %64 =   builtin Base.memoryrefget(%63, :not_atomic, false)::Any
◌  └──       goto #15
X  15 ─ %66 =   builtin Base.getfield(%3, :size)::Tuple{Int64}
X  │    %67 =   builtin Core.getfield(%66, 1)::Int64
↑′ │    %68 =   builtin Core.tuple(%64, %67)::Tuple{Any, Int64}
◌  └──       return %68
```

In order to address these difficulties, we need inference to be aware of array dimensions and propagate array dimensions in a flow-sensitive way[6], as well as come up with nice representation of loop-variant values.

`EscapeAnalysis` at this moment quickly switches to the more imprecise analysis that doesn't track precise index information in cases when array dimensions or indices are trivially non constant. The switch can naturally be implemented as a lattice join operation of `EscapeInfo.AliasInfo` property in the data-flow analysis framework.

Exception Handling

It would be also worth noting how `EscapeAnalysis` handles possible escapes via exceptions. Naively it seems enough to propagate escape information imposed on `:the_exception` object to all values that may be thrown in a corresponding `try` block. But there are actually several other ways to access to the exception object in Julia, such as `Base.current_exceptions` and `rethrow`. For example, escape analysis needs to account for potential escape of `r` in the example below:

```
julia> const GR = Ref{Any}();

julia> @noinline function rethrow_escape!()
           try
               rethrow()
           catch err
               GR[] = err
           end
       end;

julia> get′(x) = isassigned(x) ? x[] : throw(x);

julia> code_escapes() do
           r = Ref{String}()
           local t
           try
               t = get′(r)
           catch err
               t = typeof(err)   # `err` (which `r` aliases to) doesn't escape here
               rethrow_escape!() # but `r` escapes here
           end
           return t
       end
#29() in Main at REPL[4]:2
X  1 ─ %1  = %new(Base.RefValue{String})::Base.RefValue{String}
◌  2 ─ %2  = enter #8
◌  3 ─ %3  =   builtin Base.isdefined(%1, :x)::Bool
◌  └──       goto #5 if not %3
X  4 ─ %5  =   builtin Base.getfield(%1, :x)::String
◌  └──       goto #6
◌  5 ─         builtin Main.throw(%1)::Union{}
◌  └──       unreachable
◌  6 ─       $(Expr(:leave, :(%2)))
◌  7 ─       goto #9
✓  8 ┄          invoke Main.rethrow_escape!()::Any
◌  └──       $(Expr(:pop_exception, :(%2)))::Core.Const(nothing)
```

```
X  9 ┄ %13 = φ (#7 => %5, #8 => Base.RefValue{String})::Union{String,
↪  Type{Base.RefValue{String}}}
◌  └──       return %13
```

It requires a global analysis in order to correctly reason about all possible escapes via existing exception interfaces. For now we always propagate the topmost escape information to all potentially thrown objects conservatively, since such an additional analysis might not be worthwhile to do given that exception handling and error path usually don't need to be very performance sensitive, and also optimizations of error paths might be very ineffective anyway since they are often even "unoptimized" intentionally for latency reasons.

`x::EscapeInfo`'s `x.ThrownEscape` property records SSA statements where `x` can be thrown as an exception. Using this information `EscapeAnalysis` can propagate possible escapes via exceptions limitedly to only those may be thrown in each `try` region:

```
julia> result = code_escapes((String,String)) do s1, s2
           r1 = Ref(s1)
           r2 = Ref(s2)
           local ret
           try
               s1 = getˊ(r1)
               ret = sizeof(s1)
           catch err
               global GV = err # will definitely escape `r1`
           end
           s2 = getˊ(r2)       # still `r2` doesn't escape fully
           return s2
       end
#32(✓ s1::String, ✓ s2::String) in Main at REPL[1]:2
◌  1 ─     nothing::Nothing
◌  2 ─     nothing::Nothing
◌  3 ─     nothing::Nothing
◌  4 ─     nothing::Nothing
◌  5 ─     return _3
```

Analysis Usage

`analyze_escapes` is the entry point to analyze escape information of SSA-IR elements.

Most optimizations like SROA (`sroa_pass!`) are more effective when applied to an optimized source that the inlining pass (`ssa_inlining_pass!`) has simplified by resolving inter-procedural calls and expanding callee sources. Accordingly, `analyze_escapes` is also able to analyze post-inlining IR and collect escape information that is useful for certain memory-related optimizations.

However, since certain optimization passes like inlining can change control flows and eliminate dead code, they can break the inter-procedural validity of escape information. In particularity, in order to collect inter-procedurally valid escape information, we need to analyze a pre-inlining IR.

Because of this reason, `analyze_escapes` can analyze `IRCode` at any Julia-level optimization stage, and especially, it is supposed to be used at the following two stages:

- `IPO EA`: analyze pre-inlining IR to generate IPO-valid escape information cache

- `Local EA`: analyze post-inlining IR to collect locally-valid escape information

Escape information derived by `IPO EA` is transformed to the `ArgEscapeCache` data structure and cached globally. By passing an appropriate `get_escape_cache` callback to `analyze_escapes`, the escape analysis can improve analysis accuracy by utilizing cached inter-procedural information of non-inlined callees that has been derived by previous `IPO EA`. More interestingly, it is also valid to use `IPO EA` escape information for type inference, e.g., inference accuracy can be improved by forming `Const`/`PartialStruct`/`MustAlias` of mutable object.

`Base.Compiler.EscapeAnalysis.analyze_escapes` – Function.

```
analyze_escapes(ir::IRCode, nargs::Int, get_escape_cache) -> estate::EscapeState
```

Analyzes escape information in `ir`:

- `nargs`: the number of actual arguments of the analyzed call
- `get_escape_cache(::MethodInstance) -> Union{Bool,ArgEscapeCache}`: retrieves cached argument escape information

source

`Base.Compiler.EscapeAnalysis.EscapeState` – Type.

```
estate::EscapeState
```

Extended lattice that maps arguments and SSA values to escape information represented as `EscapeInfo`. Escape information imposed on SSA IR element `x` can be retrieved by `estate[x]`.

source

`Base.Compiler.EscapeAnalysis.EscapeInfo` – Type.

```
x::EscapeInfo
```

A lattice for escape information, which holds the following properties:

- `x.Analyzed::Bool`: not formally part of the lattice, only indicates whether `x` has been analyzed
- `x.ReturnEscape::Bool`: indicates `x` can escape to the caller via return
- `x.ThrownEscape::BitSet`: records SSA statement numbers where `x` can be thrown as exception:
  - `isempty(x.ThrownEscape)`: `x` will never be thrown in this call frame (the bottom)
  - `pc ∈ x.ThrownEscape`: `x` may be thrown at the SSA statement at `pc`
  - `-1 ∈ x.ThrownEscape`: `x` may be thrown at arbitrary points of this call frame (the top)

  This information will be used by `escape_exception!` to propagate potential escapes via exception.
- `x.AliasInfo::Union{Bool,IndexableFields,Unindexable}`: maintains all possible values that can be aliased to fields or array elements of `x`:
  - `x.AliasInfo === false` indicates the fields/elements of `x` aren't analyzed yet
  - `x.AliasInfo === true` indicates the fields/elements of `x` can't be analyzed, e.g. the type of `x` is not known or is not concrete and thus its fields/elements can't be known precisely

    - `x.AliasInfo::IndexableFields` records all the possible values that can be aliased to fields of object `x` with precise index information
    - `x.AliasInfo::Unindexable` records all the possible values that can be aliased to fields/elements of `x` without precise index information
- `x.Liveness::BitSet`: records SSA statement numbers where `x` should be live, e.g. to be used as a call argument, to be returned to a caller, or preserved for `:foreigncall`:
    - `isempty(x.Liveness)`: `x` is never be used in this call frame (the bottom)
    - `0 ∈ x.Liveness` also has the special meaning that it's a call argument of the currently analyzed call frame (and thus it's visible from the caller immediately).
    - `pc ∈ x.Liveness`: `x` may be used at the SSA statement at `pc`
    - `-1 ∈ x.Liveness`: `x` may be used at arbitrary points of this call frame (the top)

There are utility constructors to create common `EscapeInfo`s, e.g.,

- `NoEscape()`: the bottom(-like) element of this lattice, meaning it won't escape to anywhere
- `AllEscape()`: the topmost element of this lattice, meaning it will escape to everywhere

`analyze_escapes` will transition these elements from the bottom to the top, in the same direction as Julia's native type inference routine. An abstract state will be initialized with the bottom(-like) elements:

- the call arguments are initialized as `ArgEscape()`, whose `Liveness` property includes `0` to indicate that it is passed as a call argument and visible from a caller immediately
- the other states are initialized as `NotAnalyzed()`, which is a special lattice element that is slightly lower than `NoEscape`, but at the same time doesn't represent any meaning other than it's not analyzed yet (thus it's not formally part of the lattice)

source

---

[1] Our type inference implementation takes the alternative approach, where each lattice property is represented by a special lattice element type object. It turns out that it started to complicate implementations of the lattice operations mainly because it often requires conversion rules between each lattice element type object. And we are working on overhauling our type inference lattice implementation with `EscapeInfo`-like lattice design.

[2] *A Graph-Free approach to Data-Flow Analysis*. Markas Mohnen, 2002, April. https://api.semanticscholar.org/CorpusID:28519618.

[3] Our type inference algorithm in contrast is implemented as a forward analysis, because type information usually flows from "definition" to "usage" and it is more natural and effective to propagate such information in a forward way.

[4] In some cases, however, object fields can't be analyzed precisely. For example, object may escape to somewhere `EscapeAnalysis` can't account for possible memory effects on it, or fields of the objects simply can't be known because of the lack of type information. In such cases `AliasInfo` property is raised to the topmost element within its own lattice order, and it causes succeeding field analysis to be conservative and escape information imposed on fields of an unanalyzable object to be propagated to the object itself.

[5] *Escape Analysis in the Context of Dynamic Compilation and Deoptimization*. Thomas Kotzmann and Hanspeter Mössenböck, 2005, June. https://dl.acm.org/doi/10.1145/1064979.1064996.

[6] Otherwise we will need yet another forward data-flow analysis on top of the escape analysis.

108.26 Ahead of Time Compilation

This document describes the design and structure of the ahead-of-time (AOT) compilation system in Julia. This system is used when generating system images and package images. Much of the implementation described here is located in `aotcompile.cpp`, `staticdata.c`, and `processor.cpp`

Introduction

Though Julia normally compiles code just-in-time (JIT), it is possible to compile code ahead of time and save the resulting code to a file. This can be useful for a number of reasons:

1. To reduce the time it takes to start a Julia process.
2. To reduce the time spent in the JIT compiler instead of executing code (time to first execution, TTFX).
3. To reduce the amount of memory used by the JIT compiler.

High-Level Overview

The following descriptions are a snapshot of the current implementation details of the end-to-end pipeline that happens internally when the user compiles a new AOT module, such as occurs when they type `using Foo`. These details are likely to change over time as we implement better ways to handle them, so current implementations may not exactly match the dataflow and functions described below.

Compiling Code Images

Firstly, the methods that need to be compiled to native code must be identified. This can only be done by actually executing the code to be compiled, as the set of methods that need to be compiled depends on the types of the arguments passed to the methods, and method invocations with certain combinations of types may not be known until runtime. During this process, the exact methods that the compiler sees are tracked for later compilation, producing a compilation trace.

Note

Currently when compiling images, Julia runs the trace generation in a different process than the process performing the AOT compilation. This can have impacts when attempting to use a debugger during precompilation. The best way to debug precompilation with a debugger is to use the rr debugger, record the entire process tree, use `rr ps` to identify the relevant failing process, and then use `rr replay -p PID` to replay just the failing process.

Once the methods to be compiled have been identified, they are passed to the `jl_create_system_image` function. This function sets up a number of data structures that will be used when serializing native code to a file, and then calls `jl_create_native` with the array of methods. `jl_create_native` runs codegen on the methods produces one or more LLVM modules. `jl_create_system_image` then records some useful information about what codegen produced from the module(s).

The module(s) are then passed to `jl_dump_native`, along with the information recorded by `jl_create_system_image`. `jl_dump_native` contains the code necessary to serialize the module(s) to bitcode, object, or assembly files depending on the command-line options passed to Julia. The serialized code and information are then written to a file as an archive.

The final step is to run a system linker on the object files in the archive produced by `jl_dump_native`. Once this step is complete, a shared library containing the compiled code is produced.

Loading Code Images

When loading a code image, the shared library produced by the linker is loaded into memory. The system image data is then loaded from the shared library. This data contains information about the types, methods, and code instances that were compiled into the shared library. This data is used to restore the state of the runtime to what it was when the code image was compiled.

If the code image was compiled with multiversioning, the loader will pick the appropriate version of each function to use based on the CPU features available on the current machine.

For system images, since no other code has been loaded, the state of the runtime is now the same as it was when the code image was compiled. For package images, the environment may have changed compared to when the code was compiled, so each method must be checked against the global method table to determine if it is still valid code.

Compiling Methods

Tracing Compiled Methods

Julia has a command-line flag to record all of the methods that are compiled by the JIT compiler, `--trace-compile=filename`. When a function is compiled and this flag has a filename, Julia will print out a precompile statement to that file with the method and argument types it was called with. This therefore generates a precompile script that can be used later in the AOT compilation process. The PrecompileTools package has tooling that can make taking advantage of this functionality easier for package developers.

`jl_create_system_image`

`jl_create_system_image` saves all of the Julia-specific metadata necessary to later restore the state of the runtime. This includes data such as code instances, method instances, method tables, and type information. This function also sets up the data structures necessary to serialize the native code to a file. Finally, it calls `jl_create_native` to create one or more LLVM modules containing the native code for the methods passed to it. `jl_create_native` is responsible for running codegen on the methods passed to it.

`jl_dump_native`

`jl_dump_native` is responsible for serializing the LLVM module containing the native code to a file. In addition to the module, the system image data produced by `jl_create_system_image` is compiled as a global variable. The output of this method is bitcode, object, and/or assembly archives containing the code and system image data.

`jl_dump_native` is typically one of the larger time sinks when emitting native code, with much of the time spent in optimizing LLVM IR and emitting machine code. Therefore, this function is capable of multithreading the optimization and machine code emission steps. This multithreading is parameterized on the size of the module, but can be explicitly overridden by setting the `JULIA_IMAGE_THREADS` environment variable. The default maximum number of threads is half the number of available threads, but setting it to be lower can reduce peak memory usage during compilation.

`jl_dump_native` can also produce native code optimized for multiple architectures, when integrated with the Julia loader. This is triggered by setting the `JULIA_CPU_TARGET` environment variable and mediated by the multiversioning pass in the optimization pipeline. To make this work with multithreading, an annotation step is added before the module is split into submodules that are emitted on their own threads, and this annotation step uses information available throughout the entire module to decide what functions are cloned for different architectures. Once the annotation has happened, individual threads can emit code

for different architectures in parallel, knowing that a different submodule is guaranteed to produce the necessary functions that will be called by a cloned function.

Some other metadata about how the module was serialized is also stored in the archive, such as the number of threads used to serialize the module and the number of functions that were compiled.

Static Linking

The final step in the AOT compilation process is to run a linker on the object files in the archive produced by `jl_dump_native`. This produces a shared library containing the compiled code. This shared library can then be loaded by Julia to restore the state of the runtime. When compiling a system image, the native linker used by a C compiler is used to produce the final shared library. For package images, the LLVM linker LLD is used to provide a more consistent linking interface.

Loading Code Images

Loading the Shared Library

The first step in loading a code image is to load the shared library produced by the linker. This is done by calling `jl_dlopen` on the path to the shared library. This function is responsible for loading the shared library and resolving all of the symbols in the library.

Loading Native Code

The loader first needs to identify whether the native code that was compiled is valid for the architecture that the loader is running on. This is necessary to avoid executing instructions that older CPUs do not recognize. This is done by checking the CPU features available on the current machine against the CPU features that the code was compiled for. When multiversioning is enabled, the loader will pick the appropriate version of each function to use based on the CPU features available on the current machine. If none of the feature sets that were multiversioned, the loader will throw an error.

Part of the multiversioning pass creates a number of global arrays of all of the functions in the module. When this process is multithreaded, an array of arrays is created, which the loader reorganizes into one large array with all of the functions that were compiled for this architecture. A similar process occurs for the global variables in the module.

Setting Up Julia State

The loader then uses the global variables and functions produced from loading native code to set up Julia runtime core data structures in the current process. This setup involves adding types and methods to the Julia runtime, and making the cached native code available for use by other Julia functions and the interpreter. For package images, each method must be validated, in that the global method table's state must match the state that the package image was compiled for. In particular, if a different set of methods exists at the load time compared to compile time of the package image, the method must be invalidated and recompiled on first use. This is necessary to ensure that execution semantics remain the same regardless of if a package was precompiled or if the code was directly executed. System images do not need to perform this validation, since the global method table is empty at load time. Thus, system images have faster load times than package images.

108.27 Static analyzer annotations for GC correctness in C code

Running the analysis

The analyzer plugin that drives the analysis ships with julia. Its source code can be found in `src/clangsa`. Running it requires the clang dependency to be build. Set the `BUILD_LLVM_CLANG` variable in your Make.user in order to build an appropriate version of clang. You may also want to use the prebuilt binaries using the `USE_BINARYBUILDER_LLVM` options.

Alternatively (or if these do not suffice), try

```
make -C src install-analysis-deps
```

from Julia's toplevel directory.

Afterwards, running the analysis over the source tree is as simple as running `make -C src analyzegc`.

General Overview

Since Julia's GC is precise, it needs to maintain correct rooting information for any value that may be referenced at any time GC may occur. These places are known as `safepoints` and in the function local context, we extend this designation to any function call that may recursively end up at a safepoint.

In generated code, this is taken care of automatically by the GC root placement pass (see the chapter on GC rooting in the LLVM codegen devdocs). However, in C code, we need to inform the runtime of any GC roots manually. This is done using the following macros:

```
// The value assigned to any slot passed as an argument to these
// is rooted for the duration of this GC frame.
JL_GC_PUSH{1,...,6}(args...)
// The values assigned into the size `n` array `rts` are rooted
// for the duration of this GC frame.
JL_GC_PUSHARGS(rts, n)
// Pop a GC frame
JL_GC_POP
```

If these macros are not used where they need to be, or they are used incorrectly, the result is silent memory corruption. As such it is very important that they are placed correctly in all applicable code.

As such, we employ static analysis (and in particular the clang static analyzer) to help ensure that these macros are used correctly. The remainder of this document gives an overview of this static analysis and describes the support needed in the julia code base to make things work.

GC Invariants

There are two simple invariants for correctness:

- All `GC_PUSH` calls need to be followed by an appropriate `GC_POP` (in practice we enforce this at the function level)
- If a value was previously not rooted at any safepoint, it may no longer be referenced afterwards

Of course the devil is in the details here. In particular to satisfy the second of the above conditions, we need to know:

- Which calls are safepoints and which are not
- Which values are rooted at any given safepoint and which are not
- When is a value referenced

For the second point in particular, we need to know which memory locations will be considered rooting at runtime (i.e. values assigned to such locations are rooted). This includes locations explicitly designated as such by passing them to one of the `GC_PUSH` macros, globally rooted locations and values, as well as any location recursively reachable from one of those locations.

Static Analysis Algorithm

The idea itself is very simple, although the implementation is quite a bit more complicated (mainly due to a large number of special cases and intricacies of C and C++). In essence, we keep track of all locations that are rooting, all values that are rootable and any expression (assignments, allocations, etc) affect the rootedness of any rootable values. Then, at any safepoint, we perform a "symbolic GC" and poison any values that are not rooted at said location. If these values are later referenced, we emit an error.

The clang static analyzer works by constructing a graph of states and exploring this graph for sources of errors. Several nodes in this graph are generated by the analyzer itself (e.g. for control flow), but the definitions above augment this graph with our own state.

The static analyzer is interprocedural and can analyze control flow across function boundaries. However, the static analyzer is not fully recursive and makes heuristic decisions about which calls to explore (additionally some calls are cross-translation unit and invisible to the analyzer). In our case, our definition of correctness requires total information. As such, we need to annotate the prototypes of all function calls with whatever information the analysis required, even if that information would otherwise be available by interprocedural static analysis.

Luckily however, we can still use this interprocedural analysis to ensure that the annotations we place on a given function are indeed correct given the implementation of said function.

The analyzer annotations

These annotations are found in src/support/analyzer_annotations.h. They are only active when the analyzer is being used and expand either to nothing (for prototype annotations) or to no-ops (for function like annotations).

`JL_NOTSAFEPOINT`

This is perhaps the most common annotation, and should be placed on any function that is known not to possibly lead to reaching a GC safepoint. In general, it is only safe for such a function to perform arithmetic, memory accesses and calls to functions either annotated `JL_NOTSAFEPOINT` or otherwise known not to be safepoints (e.g. function in the C standard library, which are hardcoded as such in the analyzer)

It is valid to keep values unrooted across calls to any function annotated with this attribute:

Usage Example:

```
void jl_get_one() JL_NOTSAFEPOINT {
  return 1;
}

jl_value_t *example() {
  jl_value_t *val = jl_alloc_whatever();
  // This is valid, even though `val` is unrooted, because
  // jl_get_one is not a safepoint
  jl_get_one();
  return val;
}
```

JL_MAYBE_UNROOTED/JL_ROOTS_TEMPORARILY

When JL_MAYBE_UNROOTED is annotated as an argument on a function, indicates that said argument may be passed, even if it is not rooted. In the ordinary course of events, the julia ABI guarantees that callers root values before passing them to callees. However, some functions do not follow this ABI and allow values to be passed to them even though they are not rooted. Note however, that this does not automatically imply that said argument will be preserved. The ROOTS_TEMPORARILY annotation provides the stronger guarantee that, not only may the value be unrooted when passed, it will also be preserved across any internal safepoints by the callee.

Note that JL_NOTSAFEPOINT essentially implies JL_MAYBE_UNROOTED/JL_ROOTS_TEMPORARILY, because the rootedness of an argument is irrelevant if the function contains no safepoints.

One additional point to note is that these annotations apply on both the caller and the callee side. On the caller side, they lift rootedness restrictions that are normally required for julia ABI functions. On the callee side, they have the reverse effect of preventing these arguments from being considered implicitly rooted.

If either of these annotations is applied to the function as a whole, it applies to all arguments of the function. This should generally only be necessary for varargs functions.

Usage example:

```
JL_DLLEXPORT void JL_NORETURN jl_throw(jl_value_t *e JL_MAYBE_UNROOTED);
jl_value_t *jl_alloc_error();

void example() {
  // The return value of the allocation is unrooted. This would normally
  // be an error, but is allowed because of the above annotation.
  jl_throw(jl_alloc_error());
}
```

JL_PROPAGATES_ROOT

This annotation is commonly found on accessor functions that return one rootable object stored within another. When annotated on a function argument, it tells the analyzer that the root for that argument also applies to the value returned by the function.

Usage Example:

```
jl_value_t *jl_svecref(jl_svec_t *t JL_PROPAGATES_ROOT, size_t i) JL_NOTSAFEPOINT;
```

```
size_t example(jl_svec_t *svec) {
  jl_value_t *val = jl_svecref(svec, 1)
  // This is valid, because, as annotated by the PROPAGATES_ROOT annotation,
  // jl_svecref propagates the rooted-ness from `svec` to `val`
  jl_gc_safepoint();
  return jl_unbox_long(val);
}
```

JL_ROOTING_ARGUMENT/JL_ROOTED_ARGUMENT

This is essentially the assignment counterpart to JL_PROPAGATES_ROOT. When assigning a value to a field of another value that is already rooted, the assigned value will inherit the root of the value it is assigned into.

Usage Example:

```
void jl_svecset(void *t JL_ROOTING_ARGUMENT, size_t i, void *x JL_ROOTED_ARGUMENT)
↪  JL_NOTSAFEPOINT


size_t example(jl_svec_t *svec) {
  jl_value_t *val = jl_box_long(10000);
  jl_svecset(svec, val);
  // This is valid, because the annotations imply that the
  // jl_svecset propagates the rooted-ness from `svec` to `val`
  jl_gc_safepoint();
  return jl_unbox_long(val);
}
```

JL_GC_DISABLED

This annotation implies that this function is only called with the GC runtime-disabled. Functions of this kind are most often encountered during startup and in the GC code itself. Note that this annotation is checked against the runtime enable/disable calls, so clang will know if you lie. This is not a good way to disable processing of a given function if the GC is not actually disabled (use `ifdef __clang_analyzer__` for that if you must).

Usage example:

```
void jl_do_magic() JL_GC_DISABLED {
  // Wildly allocate here with no regard for roots
}

void example() {
  int en = jl_gc_enable(0);
  jl_do_magic();
  jl_gc_enable(en);
}
```

JL_REQUIRE_ROOTED_SLOT

This annotation requires the caller to pass in a slot that is rooted (i.e. values assigned to this slot will be rooted).

Usage example:

```
void jl_do_processing(jl_value_t **slot JL_REQUIRE_ROOTED_SLOT) {
  *slot = jl_box_long(1);
  // Ok, only, because the slot was annotated as rooting
  jl_gc_safepoint();
}

void example() {
  jl_value_t *slot = NULL;
  JL_GC_PUSH1(&slot);
  jl_do_processing(&slot);
  JL_GC_POP();
}
```

JL_GLOBALLY_ROOTED

This annotation implies that a given value is always globally rooted. It can be applied to global variable declarations, in which case it will apply to the value of those variables (or values if the declaration if for an array), or to functions, in which case it will apply to the return value of such functions (e.g. for functions that always return some private, globally rooted value).

Usage example:

```
extern JL_DLLEXPORT jl_datatype_t *jl_any_type JL_GLOBALLY_ROOTED;
jl_ast_context_t *jl_ast_ctx(fl_context_t *fl) JL_GLOBALLY_ROOTED;
```

JL_ALWAYS_LEAFTYPE

This annotations is essentially equivalent to `JL_GLOBALLY_ROOTED`, except that is should only be used if those values are globally rooted by virtue of being a leaftype. The rooting of leaftypes is a bit complicated. They are generally rooted through `cache` field of the corresponding `TypeName`, which itself is rooted by the containing module (so they're rooted as long as the containing module is ok) and we can generally assume that leaftypes are rooted where they are used, but we may refine this property in the future, so the separate annotation helps split out the reason for being globally rooted.

The analyzer also automatically detects checks for leaftype-ness and will not complain about missing GC roots on these paths.

```
JL_DLLEXPORT jl_value_t *jl_apply_array_type(jl_value_t *type, size_t dim) JL_ALWAYS_LEAFTYPE;
```

JL_GC_PROMISE_ROOTED

This is a function-like annotation. Any value passed to this annotation will be considered rooted for the scope of the current function. It is designed as an escape hatch for analyzer inadequacy or complicated situations. However, it should be used sparingly, in favor of improving the analyzer itself.

```
void example() {
  jl_value_t *val = jl_alloc_something();
  if (some_condition) {
    // We happen to know for complicated external reasons
```

```
    // that val is rooted under these conditions
    JL_GC_PROMISE_ROOTED(val);
  }
}
```

Completeness of analysis

The analyzer only looks at local information. In particular, e.g. in the PROPAGATES_ROOT case above, it assumes that such memory is only modified in ways it can see, not in any called functions (unless it happens to decide to consider them in its analysis) and not in any concurrently running threads. As such, it may miss a few problematic cases, though in practice such concurrent modification is fairly rare. Improving the analyzer to handle more such cases may be an interesting topic for future work.

108.28 Garbage Collection in Julia

Introduction

Julia has a non-moving, partially concurrent, parallel, generational and mostly precise mark-sweep collector (an interface for conservative stack scanning is provided as an option for users who wish to call Julia from C).

Allocation

Julia uses two types of allocators, the size of the allocation request determining which one is used. Objects up to 2k bytes are allocated on a per-thread free-list pool allocator, while objects larger than 2k bytes are allocated through libc malloc.

Julia's pool allocator partitions objects on different size classes, so that a memory page managed by the pool allocator (which spans 4 operating system pages on 64bit platforms) only contains objects of the same size class. Each memory page from the pool allocator is paired with some page metadata stored on per-thread lock-free lists. The page metadata contains information such as whether the page has live objects at all, number of free slots, and offsets to the first and last objects in the free-list contained in that page. These metadata are used to optimize the collection phase: a page which has no live objects at all may be returned to the operating system without any need of scanning it, for example.

While a page that has no objects may be returned to the operating system, its associated metadata is permanently allocated and may outlive the given page. As mentioned above, metadata for allocated pages are stored on per-thread lock-free lists. Metadata for free pages, however, may be stored into three separate lock-free lists depending on whether the page has been mapped but never accessed (`page_pool_clean`), or whether the page has been lazily sweeped and it's waiting to be madvised by a background GC thread (`page_pool_lazily_freed`), or whether the page has been madvised (`page_pool_freed`).

Julia's pool allocator follows a "tiered" allocation discipline. When requesting a memory page for the pool allocator, Julia will:

- Try to claim a page from `page_pool_lazily_freed`, which contains pages which were empty on the last stop-the-world phase, but not yet madvised by a concurrent sweeper GC thread.
- If it failed claiming a page from `page_pool_lazily_freed`, it will try to claim a page from `page_pool_clean`, which contains pages which were mmaped on a previous page allocation request but never accessed.

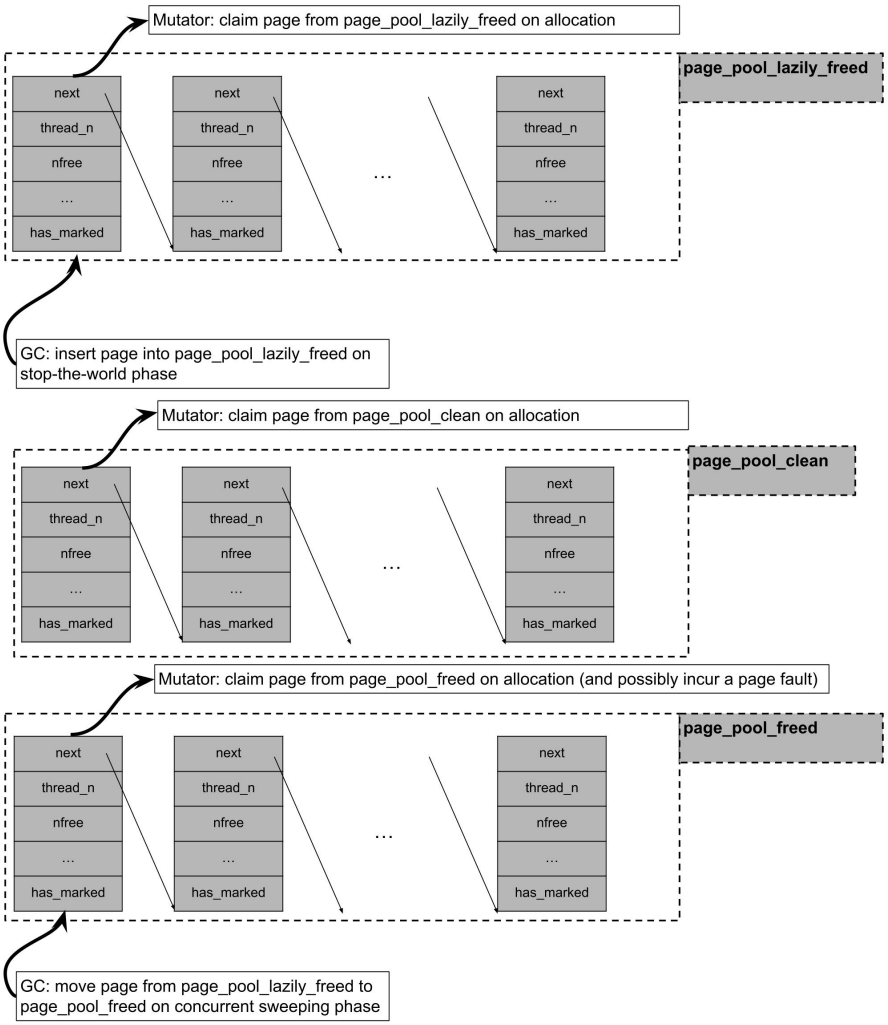


Figure 108.4: Diagram of tiered pool allocation

- If it failed claiming a page from `pool_page_clean` and from `page_pool_lazily_freed`, it will try to claim a page from `page_pool_freed`, which contains pages which have already been madvised by a concurrent sweeper GC thread and whose underlying virtual address can be recycled.
- If it failed in all of the attempts mentioned above, it will mmap a batch of pages, claim one page for itself, and insert the remaining pages into `page_pool_clean`.

Marking and Generational Collection

Julia's mark phase is implemented through a parallel iterative depth-first-search over the object graph. Julia's collector is non-moving, so object age information can't be determined through the memory region in which the object resides alone, but has to be somehow encoded in the object header or on a side table.

The lowest two bits of an object's header are used to store, respectively, a mark bit that is set when an object is scanned during the mark phase and an age bit for the generational collection.

Generational collection is implemented through sticky bits: objects are only pushed to the mark-stack, and therefore traced, if their mark-bits are not set. When objects reach the oldest generation, their mark-bits are not reset during the so-called "quick-sweep", which leads to these objects not being traced in a subsequent mark phase. A "full-sweep", however, causes the mark-bits of all objects to be reset, leading to all objects being traced in a subsequent mark phase. Objects are promoted to the next generation during every sweep phase they survive. On the mutator side, field writes are intercepted through a write barrier that pushes an object's address into a per-thread remembered set if the object is in the last generation, and if the object at the field being written is not. Objects in this remembered set are then traced during the mark phase.

Sweeping

Sweeping of object pools for Julia may fall into two categories: if a given page managed by the pool allocator contains at least one live object, then a free-list must be threaded through its dead objects; if a given page contains no live objects at all, then its underlying physical memory may be returned to the operating system through, for instance, the use of madvise system calls on Linux.

The first category of sweeping is parallelized through work-stealing. For the second category of sweeping, if concurrent page sweeping is enabled through the flag `--gcthreads=X,1` we perform the madvise system calls in a background sweeper thread, concurrently with the mutator threads. During the stop-the-world phase of the collector, pool allocated pages which contain no live objects are initially pushed into the `pool_page_lazily_freed`. The background sweeping thread is then woken up and is responsible for removing pages from `pool_page_lazily_freed`, calling madvise on them, and inserting them into `pool_page_freed`. As described above, `pool_page_lazily_freed` is also shared with mutator threads. This implies that on allocation-heavy multithreaded workloads, mutator threads would often avoid a page fault on allocation (coming from accessing a fresh mmaped page or accessing a madvised page) by directly allocating from a page in `pool_page_lazily_freed`, while the background sweeper thread needs to madvise a reduce number of pages given some of them were already claimed by the mutators.

Heuristics

GC heuristics tune the GC by changing the size of the allocation interval between garbage collections.

The GC heuristics measure how big the heap size is after a collection and set the next collection according to the algorithm described by https://dl.acm.org/doi/10.1145/3563323, in summary, it argues that the heap target should have a square root relationship with the live heap, and that it should also be scaled by how fast the GC is freeing objects and how fast the mutators are allocating. The heuristics measure the heap size by counting the number of pages that are in use and the objects that use malloc. Previously we measured the heap size by counting the alive objects, but that doesn't take into account fragmentation which could lead to bad decisions, that also meant that we used thread local information (allocations) to make decisions about a process wide (when to GC), measuring pages means the decision is global.

The GC will do full collections when the heap size reaches 80% of the maximum allowed size.

108.29 Julia + MMTk

There has been quite a lot of effort to refactor the GC code inside Julia to support external GCs. The first step to enable using different GC algorithms for Julia was the design and implementation of a GC

interface. To drive that interface, we added support for building Julia with MMTk (Memory Management Toolkit). Using Julia + MMTk enables testing different GC implementations, allowing developers to choose a specific implementation when building Julia from source. The connection between Julia and MMTk is done via a *binding*, which links the language runtime with MMTk core. The mmtk-julia binding is written in Rust and can be found in this repository.

> [!NOTE] Using a different GC requires building Julia from source. It is not possible to switch implementations at runtime. To see what version of the GC is currently being used, run `versioninfo()` from the Julia REPL and it should show the version under `GC: ...`.

Building Julia with MMTk

There are 3 different ways of building Julia with MMTk: building from source using a fixed release of the binding, checking out a custom version in the mmtk-julia repository or using a precompiled binary from Julia's BinaryBuilder. The easiest way is to use the BinaryBuilder binary. First, to enable MMTk as a third-party GC, set the variable `WITH_THIRD_PARTY_GC` to `mmtk`. Then, for example, to use the Immix as the GC, simply set the variable `MMTK_PLAN=Immix` and build Julia as usual.

There are different configurations supported by the following variables, which can be set in a `Make.user` file or as an environment variable. Note that at this time, setting `MMTK_PLAN=StickyImmix` (to use a generational version of Immix) or `MMTK_MOVING=1` (to enable object movement) will likely cause segmentation faults or other build failures, since we have not added support for these configurations yet. Setting `MMTK_BUILD=debug` will force a debug build of the binding, which will print some logging information that can be used to find errors that are specific to MMTk.

Variable		
`MMTK_PLAN`	Immix	StickyImmix
`MMTK_MOVING`	0	1
`MMTK_BUILD`	release	debug

Note that when setting only `MMTK_PLAN`, then the default is to do a non-moving, release build.

Building mmtk-julia from source

It is also possible to build the binding from source. To do so, set the variable `USE_BINARYBUILDER_MMTK_JULIA=0` and the latest release version of the binding will be downloaded and built as part of building Julia. Note that this requires an installation of the rust toolchain.

It is also possible to build a custom version of binding by checking it out from the git repository and setting a variable named `MMTK_JULIA_DIR` as the path that contains the binding.

For more information on building Julia with MMTk, please refer to the README file in the binding repo.

I've got a build error when building Julia with MMTk, what should I do?

If you try to build Julia with MMTk and get an error it is likely due to a change to Julia that has not been yet propagated to the binding or to the code in Julia that is specific to MMTk. Some changes include:

(1) **Changing the memory layout of objects in Julia**. The binding relies on automatically generated Rust FFI bindings from Julia code. These files are generated using a crate named `rust-bindgen`. To regenerate those files, check out the latest version of the `mmtk-julia` binding, set the variable `JULIA_PATH` to the path

of the Julia version you are trying to build and run `make regen-bindgen-ffi` from the directory containing the binding. This should delete the current version of the FFI bindings and generate a new version based on the Julia code from `JULIA_PATH`.

(2) **Changing the root objects passed to the GC**. Julia passes a set of objects to the GC as roots in the function gc*mark*roots. At the moment, this set needs to be consistent between both the Stock GC and MMTk (in the function `jl_gc_scan_vm_specific_roots`).

(3) **Changing how objects are scanned**. MMTk uses the same strategy to find references in Julia objects as the stock GC (see gc*mark*outrefs). Changing the logic from this function should be reflected in the Rust code in the binding that scan Julia objects.

If your case is not included in one of the alternatives above, please create an issue in the Julia repository tagging it with the `GC: MMTK` label.

108.30 JIT Design and Implementation

This document explains the design and implementation of Julia's JIT, after codegen has finished and unoptimized LLVM IR has been produced. The JIT is responsible for optimizing and compiling this IR to machine code, and for linking it into the current process and making the code available for execution.

Introduction

The JIT is responsible for managing compilation resources, looking up previously compiled code, and compiling new code. It is primarily built on LLVM's On-Request-Compilation (ORCv2) technology, which provides support for a number of useful features such as concurrent compilation, lazy compilation, and the ability to compile code in a separate process. Though LLVM provides a basic JIT compiler in the form of LLJIT, Julia uses many ORCv2 APIs directly to create its own custom JIT compiler.

Overview

Codegen produces an LLVM module containing IR for one or more Julia functions from the original Julia SSA IR produced by type inference (labeled as translate on the compiler diagram above). It also produces a mapping of code-instance to LLVM function name. However, though some optimizations have been applied by the Julia-based compiler on Julia IR, the LLVM IR produced by codegen still contains many opportunities for optimization. Thus, the first step the JIT takes is to run a target-independent optimization pipeline[1] on the LLVM module. Then, the JIT runs a target-dependent optimization pipeline, which includes target-specific optimizations and code generation, and outputs an object file. Finally, the JIT links the resulting object file into the current process and makes the code available for execution. All of this is controlled by code in `src/jitlayers.cpp`.

Currently, only one thread at a time is permitted to enter the optimize-compile-link pipeline at a time, due to restrictions imposed by one of our linkers (RuntimeDyld). However, the JIT is designed to support concurrent optimization and compilation, and the linker restriction is expected to be lifted in the future when RuntimeDyld has been fully superseded on all platforms.

[1] This is not a totally-target independent pipeline, as transformations such as vectorization rely upon target information such as vector register width and cost modeling. Additionally, codegen itself makes a few target-dependent assumptions, and the optimization pipeline will take advantage of that knowledge.

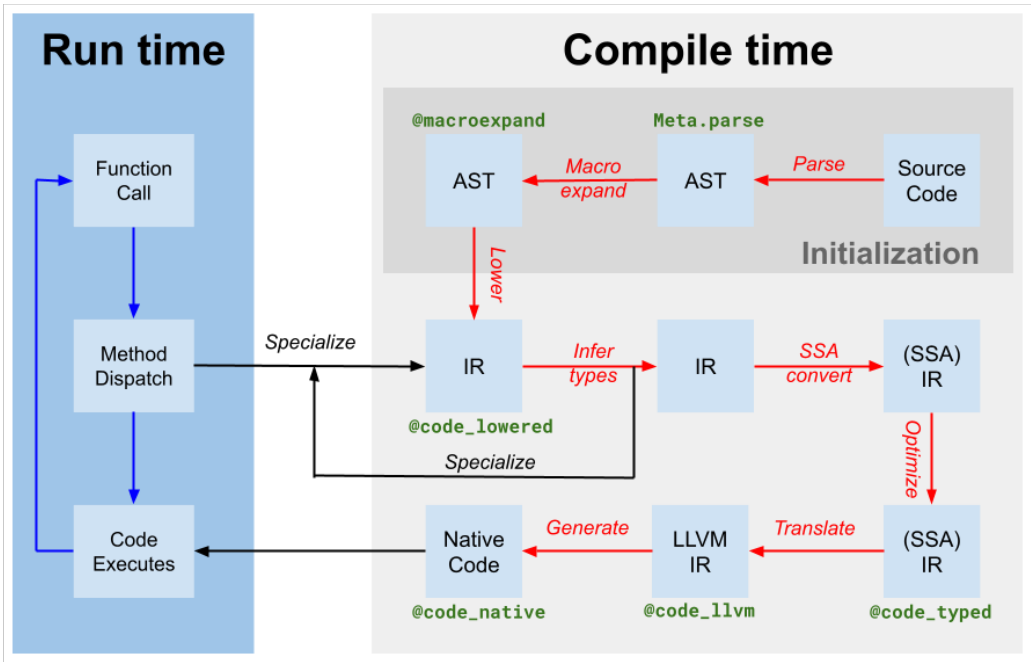


Figure 108.5: Diagram of the compiler flow

Optimization Pipeline

The optimization pipeline is based off LLVM's new pass manager, but the pipeline is customized for Julia's needs. The pipeline is defined in `src/pipeline.cpp`, and broadly proceeds through a number of stages as detailed below.

1. Early Simplification

    1. These passes are mainly used to simplify the IR and canonicalize patterns so that later passes can identify those patterns more easily. Additionally, various intrinsic calls such as branch prediction hints and annotations are lowered into other metadata or other IR features. `SimplifyCFG` (simplify control flow graph), `DCE` (dead code elimination), and `SROA` (scalar replacement of aggregates) are some of the key players here.

2. Early Optimization

    1. These passes are typically cheap and are primarily focused around reducing the number of instructions in the IR and propagating knowledge to other instructions. For example, `EarlyCSE` is used to perform common subexpression elimination, and `InstCombine` and `InstSimplify` perform a number of small peephole optimizations to make operations less expensive.

3. Loop Optimization

    1. These passes canonicalize and simplify loops. Loops are often hot code, which makes loop optimization extremely important for performance. Key players here include `LoopRotate`, `LICM`, and `LoopFullUnroll`. Some bounds check elimination also happens here, as a result of the `IRCE` pass which can prove certain bounds are never exceeded.

4. Scalar Optimization

    1. The scalar optimization pipeline contains a number of more expensive, but more powerful passes such as `GVN` (global value numbering), `SCCP` (sparse conditional constant propagation), and another round of bounds check elimination. These passes are expensive, but they can often remove large amounts of code and make vectorization much more successful and effective. Several other simplification and optimization passes intersperse the more expensive ones to reduce the amount of work they have to do.

5. Vectorization

    1. Automatic vectorization is an extremely powerful transformation for CPU-intensive code. Briefly, vectorization allows execution of a single instruction on multiple data (SIMD), e.g. performing 8 addition operations at the same time. However, proving code to be both capable of vectorization and profitable to vectorize is difficult, and this relies heavily on the prior optimization passes to massage the IR into a state where vectorization is worth it.

6. Intrinsic Lowering

    1. Julia inserts a number of custom intrinsics, for reasons such as object allocation, garbage collection, and exception handling. These intrinsics were originally placed to make optimization opportunities more obvious, but they are now lowered into LLVM IR to enable the IR to be emitted as machine code.

7. Cleanup

    1. These passes are last-chance optimizations, and perform small optimizations such as fused multiply-add propagation and division-remainder simplification. Additionally, targets that do not support half-precision floating point numbers will have their half-precision instructions lowered into single-precision instructions here, and passes are added to provide sanitizer support.

Target-Dependent Optimization and Code Generation

LLVM provides target-dependent optimization and machine code generation in the same pipeline, located in the TargetMachine for a given platform. These passes include instruction selection, instruction scheduling, register allocation, and machine code emission. The LLVM documentation provides a good overview of the process, and the LLVM source code is the best place to look for details on the pipeline and passes.

Linking

Currently, Julia is transitioning between two linkers: the older RuntimeDyld linker, and the newer JITLink linker. JITLink contains a number of features that RuntimeDyld does not have, such as concurrent and reentrant linking, but currently lacks good support for profiling integrations and does not yet support all of the platforms that RuntimeDyld supports. Over time, JITLink is expected to replace RuntimeDyld entirely. Further details on JITLink can be found in the LLVM documentation.

Execution

Once the code has been linked into the current process, it is available for execution. This fact is made known to the generating codeinst by updating the `invoke`, `specsigflags`, and `specptr` fields appropriately. Codeinsts support upgrading `invoke`, `specsigflags`, and `specptr` fields, so long as every combination of these fields that exists at any given point in time is valid to be called. This allows the JIT to update these fields without invalidating existing codeinsts, supporting a potential future concurrent JIT. Specifically, the following states may be valid:

1. `invoke` is NULL, `specsigflags` is 0b00, `specptr` is NULL
    1. This is the initial state of a codeinst, and indicates that the codeinst has not yet been compiled.
2. `invoke` is non-null, `specsigflags` is 0b00, `specptr` is NULL
    1. This indicates that the codeinst was not compiled with any specialization, and that the codeinst should be invoked directly. Note that in this instance, `invoke` does not read either the `specsigflags` or `specptr` fields, and therefore they may be modified without invalidating the `invoke` pointer.
3. `invoke` is non-null, `specsigflags` is 0b10, `specptr` is non-null
    1. This indicates that the codeinst was compiled, but a specialized function signature was deemed unnecessary by codegen.
4. `invoke` is non-null, `specsigflags` is 0b11, `specptr` is non-null
    1. This indicates that the codeinst was compiled, and a specialized function signature was deemed necessary by codegen. The `specptr` field contains a pointer to the specialized function signature. The `invoke` pointer is permitted to read both `specsigflags` and `specptr` fields.

In addition, there are a number of different transitional states that occur during the update process. To account for these potential situations, the following write and read patterns should be used when dealing with these codeinst fields.

1. When writing `invoke`, `specsigflags`, and `specptr`:
    1. Perform an atomic compare-exchange operation of specptr assuming the old value was NULL. This compare-exchange operation should have at least acquire-release ordering, to provide ordering guarantees of the remaining memory operations in the write.
    2. If `specptr` was non-null, cease the write operation and wait for bit 0b10 of `specsigflags` to be written, then restart from step 1 if desired.
    3. Write the new low bit of `specsigflags` to its final value. This may be a relaxed write.
    4. Write the new `invoke` pointer to its final value. This must have at least a release memory ordering to synchronize with reads of `invoke`.
    5. Set the second bit of `specsigflags` to 1. This must be at least a release memory ordering to synchronize with reads of `specsigflags`. This step completes the write operation and announces to all other threads that all fields have been set.

2. When reading all of `invoke`, `specsigflags`, and `specptr`:

    1. Read the `specptr` field with any memory ordering.
    2. Read the `invoke` field with at least an acquire memory ordering. This load will be referred to as `initial_invoke`.
    3. If `initial_invoke` is NULL, the codeinst is not yet executable. `invoke` is NULL, `specsigflags` may be treated as 0b00, `specptr` may be treated as NULL.
    4. If `specptr` is NULL, then the `initial_invoke` pointer must not be relying on `specptr` to guarantee correct execution. Therefore, `invoke` is non-null, `specsigflags` may be treated as 0b00, `specptr` may be treated as NULL.
    5. If `specptr` is non-null, then `initial_invoke` might not be the final `invoke` field that uses `specptr`. This can occur if `specptr` has been written, but `invoke` has not yet been written. Therefore, spin on the second bit of `specsigflags` until it is set to 1 with at least acquire memory ordering.
    6. Re-read the `invoke` field with any memory ordering. This load will be referred to as `final_invoke`.
    7. Read the `specsigflags` field with any memory ordering.
    8. `invoke` is `final_invoke`, `specsigflags` is the value read in step 7, `specptr` is the value read in step 3.

3. When updating a `specptr` to a different but equivalent function pointer:

    1. Perform a release store of the new function pointer to `specptr`. Races here must be benign, as the old function pointer is required to still be valid, and any new ones are also required to be valid as well. Once a pointer has been written to `specptr`, it must always be callable whether or not it is later overwritten.

Correctly reading these fields is implemented in `jl_read_codeinst_invoke`.

Although these write, read, and update steps are complicated, they ensure that the JIT can update codeinsts without invalidating existing codeinsts, and that the JIT can update codeinsts without invalidating existing `invoke` pointers. This allows the JIT to potentially reoptimize functions at higher optimization levels in the future, and also will allow the JIT to support concurrent compilation of functions in the future.

108.31 Core.Builtins

The following builtin functions are considered unstable, but provide the basic definitions for what defines the abilities and behaviors of a Julia program. They are typically accessed through a higher level generic API.

Raw access to memory

`Core.Intrinsics.pointerref` – Function.

```
Core.Intrinsics.pointerref(p::Ptr{T}, i::Int, align::Int)
```

Load a value of type T from the address of the `i`th element (1-indexed) starting at `p`. This is equivalent to the C expression `p[i-1]`.

The alignment must be a power of two, or 0, indicating the default alignment for T. If `p[i-1]` is out of bounds, invalid, or is not aligned, the behavior is undefined. An alignment of 1 is always safe.

See also `unsafe_load`.

source

`Core.Intrinsics.pointerset` – Function.

```
Core.Intrinsics.pointerset(p::Ptr{T}, x::T, i::Int, align::Int)
```

Store a value of type T to the address of the `i`th element (1-indexed) starting at `p`. This is equivalent to the C expression `p[i-1] = x`.

The alignment must be a power of two, or `0`, indicating the default alignment for T. If `p[i-1]` is out of bounds, invalid, or is not aligned, the behavior is undefined. An alignment of 1 is always safe.

See also `unsafe_store!`.

source

`Core.Intrinsics.atomic_pointerref` – Function.

```
Core.Intrinsics.atomic_pointerref(pointer::Ptr{T}, order::Symbol) --> T
```

Julia 1.7

This function requires Julia 1.7 or later.

See `unsafe_load`.

source

`Core.Intrinsics.atomic_pointerset` – Function.

```
Core.Intrinsics.atomic_pointerset(pointer::Ptr{T}, new::T, order::Symbol) --> pointer
```

Julia 1.7

This function requires Julia 1.7 or later.

See `unsafe_store!`.

source

`Core.Intrinsics.atomic_pointerswap` – Function.

```
Core.Intrinsics.atomic_pointerswap(pointer::Ptr{T}, new::T, order::Symbol) --> old
```

Julia 1.7

This function requires Julia 1.7 or later.

See `unsafe_swap!`.

source

`Core.Intrinsics.atomic_pointermodify` – Function.

```
Core.Intrinsics.atomic_pointermodify(pointer::Ptr{T}, function::(old::T,arg::S)->T, arg::S,
↪  order::Symbol) --> old
```

Julia 1.7

This function requires Julia 1.7 or later.

See `unsafe_modify!`.

source

`Core.Intrinsics.atomic_pointerreplace` – Function.

```
Core.Intrinsics.atomic_pointerreplace(pointer::Ptr{T}, expected::Any, new::T,
↪  success_order::Symbol, failure_order::Symbol) --> (old, cmp)
```

Julia 1.7

This function requires Julia 1.7 or later.

See `unsafe_replace!`.

source

Managed memory

`Core.memorynew` – Function.

```
Core.memorynew(::Type{T} where T <: GenericMemory, n::Int)
```

Construct an uninitialized `GenericMemory` of length n.

See also `Memory`, `Memory{T}(undef, n)`.

source

`Core.memoryrefnew` – Function.

```
Core.memoryrefnew(::GenericMemory)
Core.memoryrefnew(::GenericMemoryRef, index::Int, [boundscheck::Bool])
```

Return a `GenericMemoryRef` for a `GenericMemory`. See `memoryref`.

Julia 1.11

This function requires Julia 1.11 or later.

source

`Core.memoryrefoffset` – Function.

```
Core..memoryrefoffset(::GenericMemoryRef)
```

Return the offset index that was used to construct the `MemoryRef`. See `memoryref`.

Julia 1.11

This function requires Julia 1.11 or later.

source

`Core.memoryrefget` – Function.

```
Core.memoryrefget(::GenericMemoryRef, ordering::Symbol, boundscheck::Bool)
```

Return the value stored at the `MemoryRef`, throwing a `BoundsError` if the `Memory` is empty. See `ref[]`. The memory ordering specified must be compatible with the `isatomic` parameter.

Julia 1.11

This function requires Julia 1.11 or later.

source

`Core.memoryrefset!` – Function.

```
Core.memoryrefset!(::GenericMemoryRef, value, ordering::Symbol, boundscheck::Bool)
```

Store the value to the `MemoryRef`, throwing a `BoundsError` if the `Memory` is empty. See `ref[] = value`. The memory ordering specified must be compatible with the `isatomic` parameter.

Julia 1.11

This function requires Julia 1.11 or later.

source

`Core.memoryref_isassigned` – Function.

```
Core.memoryref_isassigned(::GenericMemoryRef, ordering::Symbol, boundscheck::Bool)
```

Return whether there is a value stored at the `MemoryRef`, returning false if the `Memory` is empty. See `isassigned(::Base.RefValue)`, `Core.memoryrefget`. The memory ordering specified must be compatible with the `isatomic` parameter.

Julia 1.11

This function requires Julia 1.11 or later.

source

`Core.memoryrefswap!` – Function.

```
Core.memoryrefswap!(::GenericMemoryRef, value, ordering::Symbol, boundscheck::Bool)
```

Atomically perform the operations to simultaneously get and set a `MemoryRef` value.

Julia 1.11

This function requires Julia 1.11 or later.

See also `swapproperty!` and `Core.memoryrefset!`.

source

`Core.memoryrefmodify!` – Function.

```
Core.memoryrefmodify!(::GenericMemoryRef, op, value, ordering::Symbol,
↪  boundscheck::Bool)::Pair
```

Atomically perform the operations to get and set a `MemoryRef` value after applying the function `op`.

Julia 1.11

This function requires Julia 1.11 or later.

See also `modifyproperty!` and `Core.memoryrefset!`.

source

`Core.memoryrefreplace!` – Function.

```
Core.memoryrefreplace!(::GenericMemoryRef, expected, desired,
                       success_order::Symbol, fail_order::Symbol=success_order,
                       ↪  boundscheck::Bool) -> (; old, success::Bool)
```

Atomically perform the operations to get and conditionally set a `MemoryRef` value.

Julia 1.11

This function requires Julia 1.11 or later.

See also `replaceproperty!` and `Core.memoryrefset!`.

source

`Core.memoryrefsetonce!` – Function.

```
Core.memoryrefsetonce!(::GenericMemoryRef, value,
                       success_order::Symbol, fail_order::Symbol=success_order,
                       ↪  boundscheck::Bool) -> success::Bool
```

Atomically perform the operations to set a `MemoryRef` to a given value, only if it was previously not set.

Julia 1.11

This function requires Julia 1.11 or later.

See also `setpropertyonce!` and `Core.memoryrefset!`.

source

Module bindings

`Core.get_binding_type` – Function.

```
Core.get_binding_type(module::Module, name::Symbol)
```

Retrieve the declared type of the binding `name` from the module `module`.

Julia 1.9

This function requires Julia 1.9 or later.

source

Other

`Core.IntrinsicFunction` – Type.

```
Core.IntrinsicFunction <: Core.Builtin <: Function
```

The `Core.IntrinsicFunction` function define some basic primitives for what defines the abilities and behaviors of a Julia program

source

`Core.Intrinsics` – Module.

```
Core.Intrinsics
```

The `Core.Intrinsics` module holds the `Core.IntrinsicFunction` objects.

source

`Core.IR` – Module.

```
Core.IR
```

The `Core.IR` module exports the IR object model.

source

`Base.quoted` – Function.

```
quoted(x)
```

Return x made safe for inserting as a constant into IR. Note that this does not make it safe for inserting into an AST, since eval will sometimes copy some types of AST object inside, and even may sometimes evaluate and interpolate any $ inside, depending on the context.

source

`Base.isa_ast_node` – Function.

```
isa_ast_node(x)
```

Return false if x is not interpreted specially by any of inference, lowering, or codegen as either an AST or IR special form.

source

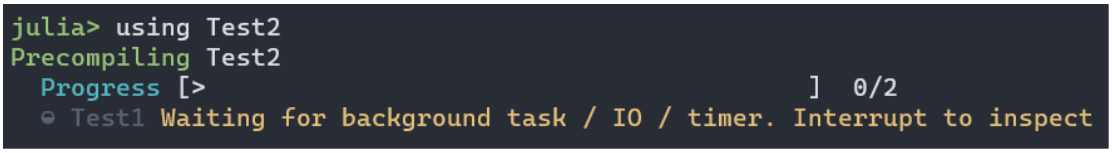


Figure 108.6: Screenshot of precompilation hang

108.32 Fixing precompilation hangs due to open tasks or IO

On Julia 1.10 or higher, you might see the following message:

This may repeat. If it continues to repeat with no hints that it will resolve itself, you may have a "precompilation hang" that requires fixing. Even if it's transient, you might prefer to resolve it so that users will not be bothered by this warning. This page walks you through how to analyze and fix such issues.

If you follow the advice and hit `Ctrl-C`, you might see

```
^C Interrupted: Exiting precompilation...

  1 dependency had warnings during precompilation:
┌ Test1 [ac89d554-e2ba-40bc-bc5c-de68b658c982]
│  [pid 2745] Waiting for background task / IO / timer to finish:
│   Handle type        uv_handle_t->data
│   timer              0x55580decd1e0->0x7f94c3a4c340
```

This message conveys two key pieces of information:

- the hang is occurring during precompilation of `Test1`, a dependency of `Test2` (the package we were trying to load with `using Test2`)
- during precompilation of `Test1`, Julia created a `Timer` object (use `?Timer` if you're unfamiliar with Timers) which is still open; until that closes, the process is hung

If this is enough of a hint for you to figure out how `timer = Timer(args...)` is being created, one good solution is to add `wait(timer)` if `timer` eventually finishes on its own, or `close(timer)` if you need to force-close it, before the final end of the module.

However, there are cases that may not be that straightforward. Usually the best option is to start by determining whether the hang is due to code in Test1 or whether it is due to one of Test1's dependencies:

- Option 1: `Pkg.add("Aqua")` and use `Aqua.test_persistent_tasks`. This should help you identify which package is causing the problem, after which the instructions below should be followed. If needed, you can create a `PkgId` as `Base.PkgId(UUID("..."), "Test1")`, where `...` comes from the `uuid` entry in `Test1/Project.toml`.
- Option 2: manually diagnose the source of the hang.

To manually diagnose:

1. `Pkg.develop("Test1")`
2. Comment out all the code `included` or defined in `Test1`, *except* the `using`/`import` statements.
3. Try `using Test2` (or even `using Test1` assuming that hangs too) again

Now we arrive at a fork in the road: either

- the hang persists, indicating it is due to one of your dependencies
- the hang disappears, indicating that it is due to something in your code.

Diagnosing and fixing hangs due to a package dependency

Use a binary search to identify the problematic dependency: start by commenting out half your dependencies, then when you isolate which half is responsible comment out half of that half, etc. (You don't have to remove them from the project, just comment out the `using`/`import` statements.)

Once you've identified a suspect (here we'll call it `ThePackageYouThinkIsCausingTheProblem`), first try precompiling that package. If it also hangs during precompilation, continue chasing the problem backwards.

However, most likely `ThePackageYouThinkIsCausingTheProblem` will precompile fine. This suggests it's in the function `ThePackageYouThinkIsCausingTheProblem.__init__`, which does not run during precompilation of `ThePackageYouThinkIsCausingTheProblem` but *does* in any package that loads `ThePackageYouThinkIsCausingTheProb`
To test this theory, set up a minimal working example (MWE), something like

```
(@v1.10) pkg> generate MWE
  Generating  project MWE:
    MWE\Project.toml
    MWE\src\MWE.jl
```

where the source code of `MWE.jl` is

```
module MWE
using ThePackageYouThinkIsCausingTheProblem
end
```

and you've added `ThePackageYouThinkIsCausingTheProblem` to MWE's dependencies.

If that MWE reproduces the hang, you've found your culprit: `ThePackageYouThinkIsCausingTheProblem.__init__` must be creating the `Timer` object. If the timer object can be safely `closed`, that's a good option. Otherwise, the most common solution is to avoid creating the timer while *any* package is being precompiled: add

```
ccall(:jl_generating_output, Cint, ()) == 1 && return nothing
```

as the first line of `ThePackageYouThinkIsCausingTheProblem.__init__`, and it will avoid doing any initialization in any Julia process whose purpose is to precompile packages.

Fixing package code to avoid hangs

Search your package for suggestive words (here like "Timer") and see if you can identify where the problem is being created. Note that a method *definition* like

```
maketimer() = Timer(timer -> println("hi"), 0; interval=1)
```

is not problematic in and of itself: it can cause this problem only if `maketimer` gets called while the module is being defined. This might be happening from a top-level statement such as

```
const GLOBAL_TIMER = maketimer()
```

or it might conceivably occur in a precompile workload.

If you struggle to identify the causative lines, then consider doing a binary search: comment out sections of your package (or `include` lines to omit entire files) until you've reduced the problem in scope.

Chapter 109

Developing/debugging Julia's C code

109.1 Reporting and analyzing crashes (segfaults)

So you managed to break Julia. Congratulations! Collected here are some general procedures you can undergo for common symptoms encountered when something goes awry. Including the information from these debugging steps can greatly help the maintainers when tracking down a segfault or trying to figure out why your script is running slower than expected.

If you've been directed to this page, find the symptom that best matches what you're experiencing and follow the instructions to generate the debugging information requested. Table of symptoms:

- Segfaults during bootstrap (`sysimg.jl`)
- Segfaults when running a script
- Errors during Julia startup
- Other generic segfaults or unreachables reached

Version/Environment info

No matter the error, we will always need to know what version of Julia you are running. When Julia first starts up, a header is printed out with a version number and date. Please also include the output of `versioninfo()` (exported from the `InteractiveUtils` standard library) in any report you create:

```
julia> using InteractiveUtils

julia> versioninfo()
Julia Version 1.13.0-rc4
Commit 8f33e09afea (2026-09-02 06:10 UTC)
Build Info:
  Official https://julialang.org release
Platform Info:
  OS: Linux (x86_64-linux-gnu)
  CPU: 4 × Intel(R) Xeon(R) Platinum 8370C CPU @ 2.80GHz
  WORD_SIZE: 64
  LLVM: libLLVM-20.1.8 (ORCJIT, icelake-server)
  GC: Built with stock GC
```

```
Threads: 1 default, 1 interactive, 1 GC (on 4 virtual cores)
Environment:
  JULIA_DOCS =
  ↪  /home/runner/work/docs.julialang.org/docs.julialang.org/pdf/build/docs.julialang.org
  JULIA_SOURCE = /home/runner/work/docs.julialang.org/docs.julialang.org/pdf/build/julia
  JULIA_EXECUTABLE =
  ↪  /home/runner/work/docs.julialang.org/docs.julialang.org/pdf/build/julia-1.13.0-rc4-linux-x86_64/bin/julia
```

Segfaults during bootstrap (`sysimg.jl`)

Segfaults toward the end of the `make` process of building Julia are a common symptom of something going wrong while Julia is preparsing the corpus of code in the `base/` folder. Many factors can contribute toward this process dying unexpectedly, however it is as often as not due to an error in the C-code portion of Julia, and as such must typically be debugged with a debug build inside of `gdb`. Explicitly:

Create a debug build of Julia:

```
$ cd <julia_root>
$ make debug
```

Note that this process will likely fail with the same error as a normal `make` incantation, however this will create a debug executable that will offer `gdb` the debugging symbols needed to get accurate backtraces. Next, manually run the bootstrap process inside of `gdb`:

```
$ cd base/
$ gdb -x ../contrib/debug_bootstrap.gdb
```

This will start `gdb`, attempt to run the bootstrap process using the debug build of Julia, and print out a backtrace if (when) it segfaults. You may need to hit `<enter>` a few times to get the full backtrace. Create a gist with the backtrace, the version info, and any other pertinent information you can think of and open a new issue on Github with a link to the gist.

Segfaults when running a script

The procedure is very similar to Segfaults during bootstrap (`sysimg.jl`). Create a debug build of Julia, and run your script inside of a debugged Julia process:

```
$ cd <julia_root>
$ make debug
$ gdb --args usr/bin/julia-debug <path_to_your_script>
```

Note that `gdb` will sit there, waiting for instructions. Type `r` to run the process, and `bt` to generate a backtrace once it segfaults:

```
(gdb) r
Starting program: /home/sabae/src/julia/usr/bin/julia-debug ./test.jl
...
(gdb) bt
```

Create a gist with the backtrace, the version info, and any other pertinent information you can think of and open a new issue on Github with a link to the gist.

Errors during Julia startup

Occasionally errors occur during Julia's startup process (especially when using binary distributions, as opposed to compiling from source) such as the following:

```
$ julia
exec: error -5
```

These errors typically indicate something is not getting loaded properly very early on in the bootup phase, and our best bet in determining what's going wrong is to use external tools to audit the disk activity of the `julia` process:

- On Linux, use `strace`:

  ```
  $ strace julia
  ```

- On OSX, use `dtruss`:

  ```
  $ dtruss -f julia
  ```

Create a gist with the `strace`/ `dtruss` output, the version info, and any other pertinent information and open a new issue on Github with a link to the gist.

Other generic segfaults or unreachables reached

As mentioned elsewhere, `julia` has good integration with `rr` for generating traces; this includes, on Linux, the ability to automatically run `julia` under `rr` and share the trace after a crash. This can be immensely helpful when debugging such crashes and is strongly encouraged when reporting crash issues to the JuliaLang/julia repo. To run `julia` under `rr` automatically, do:

```
julia --bug-report=rr
```

To generate the `rr` trace locally, but not share, you can do:

```
julia --bug-report=rr-local
```

Note that this is only works on Linux. The blog post on Time Travelling Bug Reporting has many more details.

Glossary

A few terms have been used as shorthand in this guide:

- `<julia_root>` refers to the root directory of the Julia source tree; e.g. it should contain folders such as `base`, `deps`, `src`, `test`, etc.

109.2 gdb debugging tips

Displaying Julia variables

Within gdb, any `jl_value_t*` object `obj` can be displayed using

```
(gdb) call jl_(obj)
```

The object will be displayed in the `julia` session, not in the gdb session. This is a useful way to discover the types and values of objects being manipulated by Julia's C code.

Similarly, if you're debugging some of Julia's internals (e.g., `compiler.jl`), you can print `obj` using

```
ccall(:jl_, Cvoid, (Any,), obj)
```

This is a good way to circumvent problems that arise from the order in which julia's output streams are initialized.

Julia's flisp interpreter uses `value_t` objects; these can be displayed with `call fl_print(fl_ctx, ios_stdout, obj)`.

Useful Julia variables for Inspecting

While the addresses of many variables, like singletons, can be useful to print for many failures, there are a number of additional variables (see `julia.h` for a complete list) that are even more useful.

- (when in `jl_apply_generic`) `mfunc` and `jl_uncompress_ast(mfunc->def, mfunc->code)` :: for figuring out a bit about the call-stack
- `jl_lineno` and `jl_filename` :: for figuring out what line in a test to go start debugging from (or figure out how far into a file has been parsed)
- `$1` :: not really a variable, but still a useful shorthand for referring to the result of the last gdb command (such as `print`)
- `jl_options` :: sometimes useful, since it lists all of the command line options that were successfully parsed
- `jl_uv_stderr` :: because who doesn't like to be able to interact with stdio

Useful Julia functions for Inspecting those variables

- `jl_print_task_backtraces(0)` :: Similar to gdb's `thread apply all bt` or lldb's `thread backtrace all`. Runs all threads while printing backtraces for all existing tasks.
- `jl_gdblookup($pc)` :: For looking up the current function and line.
- `jl_gdblookupinfo($pc)` :: For looking up the current method instance object.
- `jl_gdbdumpcode(mi)` :: For dumping all of `code_typed/code_llvm/code_asm` when the REPL is not working right.

- `jlbacktrace()` :: For dumping the current Julia backtrace stack to stderr. Only usable after `record_backtrace()` has been called.
- `jl_dump_llvm_value(Value*)` :: For invoking `Value->dump()` in gdb, where it doesn't work natively. For example, `f->linfo->functionObject`, `f->linfo->specFunctionObject`, and `to_function(f->linfo)`.
- `jl_dump_llvm_module(Module*)` :: For invoking `Module->dump()` in gdb, where it doesn't work natively.
- `Type->dump()` :: only works in lldb. Note: add something like `;1` to prevent lldb from printing its prompt over the output
- `jl_eval_string("expr")` :: for invoking side-effects to modify the current state or to lookup symbols
- `jl_typeof(jl_value_t*)` :: for extracting the type tag of a Julia value (in gdb, call `macro define jl_typeof jl_typeof` first, or pick something short like `ty` for the first arg to define a shorthand)

Inserting breakpoints for inspection from gdb

In your gdb session, set a breakpoint in `jl_breakpoint` like so:

```
(gdb) break jl_breakpoint
```

Then within your Julia code, insert a call to `jl_breakpoint` by adding

```
ccall(:jl_breakpoint, Cvoid, (Any,), obj)
```

where `obj` can be any variable or tuple you want to be accessible in the breakpoint.

It's particularly helpful to back up to the `jl_apply` frame, from which you can display the arguments to a function using, e.g.,

```
(gdb) call jl_(args[0])
```

Another useful frame is `to_function(jl_method_instance_t *li, bool cstyle)`. The `jl_method_instance_t*` argument is a struct with a reference to the final AST sent into the compiler. However, the AST at this point will usually be compressed; to view the AST, call `jl_uncompress_ast` and then pass the result to `jl_`:

```
#2  0x00007ffff7928bf7 in to_function (li=0x2812060, cstyle=false) at codegen.cpp:584
584         abort();
(gdb) p jl_(jl_uncompress_ast(li, li->ast))
```

Inserting breakpoints upon certain conditions

Loading a particular file

Let's say the file is `sysimg.jl`:

```
(gdb) break jl_load if strcmp(fname, "sysimg.jl")==0
```

Calling a particular method

```
(gdb) break jl_apply_generic if strcmp((char*)(jl_symbol_name)(jl_gf_mtable(F)->name),
↪  "method_to_break")==0
```

Since this function is used for every call, you will make everything 1000x slower if you do this.

Dealing with signals

Julia requires a few signals to function properly. The profiler uses `SIGUSR2` for sampling and the garbage collector uses `SIGSEGV` for threads synchronization. If you are debugging some code that uses the profiler or multiple threads, you may want to let the debugger ignore these signals since they can be triggered very often during normal operations. The command to do this in GDB is (replace `SIGSEGV` with `SIGUSR2` or other signals you want to ignore):

```
(gdb) handle SIGSEGV noprint nostop pass
```

The corresponding LLDB command is (after the process is started):

```
(lldb) pro hand -p true -s false -n false SIGSEGV
```

If you are debugging a segfault with threaded code, you can set a breakpoint on `jl_fprint_critical_error` (`sigdie_handler` should also work on Linux and BSD) in order to only catch the actual segfault rather than the GC synchronization points.

Debugging during Julia's build process (bootstrap)

Errors that occur during `make` need special handling. Julia is built in two stages, constructing `sys0` and `sys.ji`. To see what commands are running at the time of failure, use `make VERBOSE=1`.

At the time of this writing, you can debug build errors during the `sys0` phase from the `base` directory using:

```
julia/base$ gdb --args ../usr/bin/julia-debug -C native --build ../usr/lib/julia/sys0 sysimg.jl
```

You might need to delete all the files in `usr/lib/julia/` to get this to work.

You can debug the `sys.ji` phase using:

```
julia/base$ gdb --args ../usr/bin/julia-debug -C native --build ../usr/lib/julia/sys -J
↪  ../usr/lib/julia/sys0.ji sysimg.jl
```

By default, any errors will cause Julia to exit, even under gdb. To catch an error "in the act", set a breakpoint in `jl_error` (there are several other useful spots, for specific kinds of failures, including: `jl_too_few_args`, `jl_too_many_args`, and `jl_throw`).

Once an error is caught, a useful technique is to walk up the stack and examine the function by inspecting the related call to `jl_apply`. To take a real-world example:

```
Breakpoint 1, jl_throw (e=0x7ffdf42de400) at task.c:802
802 {
(gdb) p jl_(e)
ErrorException("auto_unbox: unable to determine argument type")
$2 = void
(gdb) bt 10
#0  jl_throw (e=0x7ffdf42de400) at task.c:802
#1  0x00007ffff65412fe in jl_error (str=0x7ffde56be000 <_j_str267> "auto_unbox:
   unable to determine argument type")
   at builtins.c:39
#2  0x00007ffde56bd01a in julia_convert_16886 ()
#3  0x00007ffff6541154 in jl_apply (f=0x7ffdf367f630, args=0x7fffffffc2b0, nargs=2) at
↪  julia.h:1281
...
```

The most recent `jl_apply` is at frame #3, so we can go back there and look at the AST for the function `julia_convert_16886`. This is the uniqued name for some method of `convert`. `f` in this frame is a `jl_value_t*`, so we can look at the type signature, if any, from the specTypes field:

```
(gdb) f 3
#3  0x00007ffff6541154 in jl_apply (f=0x7ffdf367f630, args=0x7fffffffc2b0, nargs=2) at
↪  julia.h:1281
1281            return f->fptr((jl_value_t*)f, args, nargs);
(gdb) p f->linfo->specTypes
$4 = (jl_tupletype_t *) 0x7ffdf39b1030
(gdb) p jl_( f->linfo->specTypes )
Tuple{Type{Float32}, Float64}           # <-- type signature for julia_convert_16886
```

Then, we can look at the AST for this function:

```
(gdb) p jl_( jl_uncompress_ast(f->linfo, f->linfo->ast) )
Expr(:lambda, Array{Any, 1}[:#s29, :x], Array{Any, 1}[Array{Any, 1}[], Array{Any, 1}[Array{Any,
↪  1}[:#s29, :Any, 0], Array{Any, 1}[:x, :Any, 0]], Array{Any, 1}[], 0], Expr(:body,
Expr(:line, 90, :float.jl)::Any,
Expr(:return, Expr(:call, :box, :Float32, Expr(:call, :fptrunc, :Float32,
↪  :x)::Any)::Any)::Any)::Any)::Any
```

Finally, and perhaps most usefully, we can force the function to be recompiled in order to step through the codegen process. To do this, clear the cached `functionObject` from the `jl_lambda_info_t*`:

```
(gdb) p f->linfo->functionObject
$8 = (void *) 0x1289d070
(gdb) set f->linfo->functionObject = NULL
```

Then, set a breakpoint somewhere useful (e.g. `emit_function`, `emit_expr`, `emit_call`, etc.), and run codegen:

```
(gdb) p jl_compile(f)
... # your breakpoint here
```

Debugging precompilation errors

Module precompilation spawns a separate Julia process to precompile each module. Setting a breakpoint or catching failures in a precompile worker requires attaching a debugger to the worker. The easiest approach is to set the debugger watch for new process launches matching a given name. For example:

```
(gdb) attach -w -n julia-debug
```

or:

```
(lldb) process attach -w -n julia-debug
```

Then run a script/command to start precompilation. As described earlier, use conditional breakpoints in the parent process to catch specific file-loading events and narrow the debugging window. (some operating systems may require alternative approaches, such as following each `fork` from the parent process)

Mozilla's Record and Replay Framework (rr)

Julia now works out of the box with rr, the lightweight recording and deterministic debugging framework from Mozilla. This allows you to replay the trace of an execution deterministically. The replayed execution's address spaces, register contents, syscall data etc are exactly the same in every run.

A recent version of rr (3.1.0 or higher) is required.

Reproducing concurrency bugs with rr

rr simulates a single-threaded machine by default. In order to debug concurrent code you can use `rr record --chaos` which will cause rr to simulate between one to eight cores, chosen randomly. You might therefore want to set `JULIA_NUM_THREADS=8` and rerun your code under rr until you have caught your bug.

109.3 Using Valgrind with Julia

Valgrind is a tool for memory debugging, memory leak detection, and profiling. This section describes things to keep in mind when using Valgrind to debug memory issues with Julia.

General considerations

By default, Valgrind assumes that there is no self modifying code in the programs it runs. This assumption works fine in most instances but fails miserably for a just-in-time compiler like `julia`. For this reason it is crucial to pass `--smc-check=all-non-file` to `valgrind`, else code may crash or behave unexpectedly (often in subtle ways).

In some cases, to better detect memory errors using Valgrind, it can help to compile `julia` with memory pools disabled. The compile-time flag `MEMDEBUG` disables memory pools in Julia, and `MEMDEBUG2` disables memory pools in FemtoLisp. To build `julia` with both flags, add the following line to `Make.user`:

```
CFLAGS = -DMEMDEBUG -DMEMDEBUG2
```

Another thing to note: if your program uses multiple worker processes, it is likely that you want all such worker processes to run under Valgrind, not just the parent process. To do this, pass `--trace-children=yes` to `valgrind`.

Yet another thing to note: if using `valgrind` errors with `Unable to find compatible target in system image`, try rebuilding the sysimage with target `generic` or julia with `JULIA_CPU_TARGET=generic`.

Suppressions

Valgrind will typically display spurious warnings as it runs. To reduce the number of such warnings, it helps to provide a suppressions file to Valgrind. A sample suppressions file is included in the Julia source distribution at `contrib/valgrind-julia.supp`.

The suppressions file can be used from the `julia/` source directory as follows:

```
$ valgrind --smc-check=all-non-file --suppressions=contrib/valgrind-julia.supp ./julia
↪  progname.jl
```

Any memory errors that are displayed should either be reported as bugs or contributed as additional suppressions. Note that some versions of Valgrind are shipped with insufficient default suppressions, so that may be one thing to consider before submitting any bugs.

Running the Julia test suite under Valgrind

It is possible to run the entire Julia test suite under Valgrind, but it does take quite some time (typically several hours). To do so, run the following command from the `julia/test/` directory:

```
valgrind --smc-check=all-non-file --trace-children=yes
↪  --suppressions=$PWD/../contrib/valgrind-julia.supp ../julia runtests.jl all
```

If you would like to see a report of "definite" memory leaks, pass the flags `--leak-check=full --show-leak-kinds=definite` to `valgrind` as well.

Additional spurious warnings

This section covers Valgrind warnings that cannot be added to the suppressions file yet are nonetheless safe to ignore.

Unhandled rr system calls

Valgrind will emit a warning if it encounters any of the system calls that are specific to rr, the Record and Replay Framework. In particular, a warning about an unhandled `1008` syscall will be shown when julia tries to detect whether it is running under rr:

```
--xxxxxx-- WARNING: unhandled amd64-linux syscall: 1008
--xxxxxx-- You may be able to write your own handler.
--xxxxxx-- Read the file README_MISSING_SYSCALL_OR_IOCTL.
--xxxxxx-- Nevertheless we consider this a bug.  Please report
--xxxxxx-- it at http://valgrind.org/support/bug_reports.html.
```

This issue has been reported to the Valgrind developers as they have requested.

Caveats

Valgrind currently does not support multiple rounding modes, so code that adjusts the rounding mode will behave differently when run under Valgrind.

In general, if after setting `--smc-check=all-non-file` you find that your program behaves differently when run under Valgrind, it may help to pass `--tool=none` to `valgrind` as you investigate further. This will enable the minimal Valgrind machinery but will also run much faster than when the full memory checker is enabled.

109.4 GC Debug Build Tools

Julia's garbage collector includes a suite of debugging tools that are enabled by building Julia with `WITH_GC_DEBUG_ENV=1`. These tools help developers diagnose memory-safety bugs such as missing write barriers, use-after-free errors, and incorrect GC roots.

Building with GC Debug Support

Add the following line to your `Make.user` file before building Julia:

```
WITH_GC_DEBUG_ENV=1
```

This sets the C preprocessor define `GC_DEBUG_ENV`, which enables several debugging features described below. It also automatically enables `GC_VERIFY` (see GC Verification).

To rebuild Julia after changing `Make.user`:

```
make -j
```

Note

The AddressSanitizer build configuration (`contrib/asan/Make.user.asan`) enables `WITH_GC_DEBUG_ENV=1` by default, since ASAN replaces pool allocation with `malloc`/`free` and the GC debug env controls align with that model.

Environment Variables

When Julia is built with `WITH_GC_DEBUG_ENV=1`, the following environment variables are recognized at startup.

`JULIA_GC_WAIT_FOR_DEBUGGER`

If set to any value other than `0`, the GC will pause and wait for a debugger to attach whenever a critical GC error is detected (such as a write barrier violation found by `GC_VERIFY`), instead of immediately aborting.

```
JULIA_GC_WAIT_FOR_DEBUGGER=1 ./julia myscript.jl
```

Once the process is sleeping, attach a debugger with:

```
gdb -p <PID>
```

JULIA_GC_ALLOC_POOL, JULIA_GC_ALLOC_OTHER, JULIA_GC_ALLOC_PRINT

These three variables control when the GC is triggered or when statistics are printed, based on allocation counts. They share a common format:

```
[r]<min>:<interv>:<max>
```

Field	Meaning
`min`	Number of allocations before the first trigger
`interv`	Interval between subsequent triggers
`max`	Maximum allocation count at which to trigger (default: no limit)
`r` prefix	Randomise trigger timing while preserving the same average frequency

All three fields are optional. Examples:

Setting	Effect
`1`	Trigger on every allocation
`100:10`	First trigger at allocation 100, then every 10 after that
`50:1:200`	Trigger every allocation from 50 to 200
`r1000:1000`	Trigger approximately every 1000 allocations with random jitter

JULIA_GC_ALLOC_POOL — Controls GC collection triggered by small (pool-allocated) objects. The counter increments on each pool allocation.

JULIA_GC_ALLOC_OTHER — Controls GC collection triggered by large (non-pool, `malloc`-backed) objects. The counter increments on each large allocation.

JULIA_GC_ALLOC_PRINT — Controls when allocation statistics are printed to `stderr`. On each trigger, a line like the following is emitted:

```
Allocations: 12345 (Pool: 10000; Other: 2345); GC: 7
```

Example: trigger GC on every allocation

This is the most aggressive stress-test mode. It runs a full GC cycle on every single allocation, which makes bugs far more likely to be exposed:

```
JULIA_GC_ALLOC_POOL=1 JULIA_GC_ALLOC_OTHER=1 ./julia myscript.jl
```

Example: randomised stress testing

Using the `r` prefix distributes GC triggers randomly around the given interval, which can expose bugs that only appear at specific allocation counts when using a fixed interval:

```
JULIA_GC_ALLOC_POOL=r1:1 JULIA_GC_ALLOC_OTHER=r1:1 ./julia myscript.jl
```

GC Verification (`GC_VERIFY`)

`WITH_GC_DEBUG_ENV=1` automatically enables `GC_VERIFY`. After every minor (quick) GC, a full second GC pass is run that:

1. Clears all mark bits.
2. Re-marks from all roots.
3. Checks that every object that was about to be freed is now also unreachable in the fresh mark phase.

If an object is found to be alive in the second pass but was scheduled for collection in the first pass, there is a missing write barrier. The verifier will then backtrack the object graph to identify which parent object was written without a corresponding `jl_gc_wb()` call, and print a message such as:

```
Missing write barrier found !
<parent> was written a reference to <child> that was not recorded
```

After printing the diagnostic, the process aborts (or waits for a debugger if `JULIA_GC_WAIT_FOR_DEBUGGER=1`).

Note

`GC_VERIFY` is limited to single-threaded GC. If `julia` is started with GC threads (`--gcthreads`), verification is silently skipped.

Tips for debugging write barrier violations

1. **Reproduce with GC_VERIFY enabled.** Build with `WITH_GC_DEBUG_ENV=1` and run the failing workload. The verifier will catch the violation and print which object and slot were written without a write barrier.
2. **Disable ASLR for reproducible addresses.** If you need to set hardware watchpoints on specific memory locations:

   ```
   echo 0 | sudo tee /proc/sys/kernel/randomize_va_space
   ```

   Addresses will then be stable across runs (with the same binary and environment).
3. **Use hardware watchpoints in GDB.** Once you know the address of the slot that was written incorrectly, attach GDB and set a watchpoint:

   ```
   watch *slot_addr if *slot_addr == expected_val
   ```

   This stops execution at the exact moment the write occurs.
4. **Use `rr` for deterministic replay.** `rr` records a process execution and replays it perfectly — the same instructions, the same addresses — allowing you to run the program backwards in GDB. This is especially useful for GC bugs where the failure manifests far from the root cause:

   ```
   rr record ./julia myscript.jl
   rr replay
   ```

Inside the replay session, use reverse-execution commands such as `reverse-continue` (`rc`) and `reverse-next` (`rn`) to step backwards from the crash to the moment the bad write occurred. Because `rr` replays with ASLR disabled and a fixed random seed, addresses are stable across record/replay cycles, which makes watchpoints reliable without having to manually disable ASLR.

Related Tools

- Using Valgrind with Julia — detect memory errors and leaks using `MEMDEBUG` plus Valgrind's `memcheck` tool.
- Sanitizer support — build Julia with AddressSanitizer or ThreadSanitizer; the ASAN configuration automatically sets `WITH_GC_DEBUG_ENV=1`.
- Static analyzer annotations for GC correctness in C code — static analysis tool that checks for missing `JL_GC_PUSH` / write barrier annotations in C source files.

109.5 External Profiler Support

Julia provides explicit support for some external tracing profilers, enabling you to obtain a high-level overview of the runtime's execution behavior.

The currently supported profilers are:

- Tracy
- Intel VTune (ITTAPI)
- NVIDIA Nsight Systems (NVTX)

Adding New Zones

From C/C++ code

To add new zones, use the `JL_TIMING` macro. You can find numerous examples throughout the codebase by searching for `JL_TIMING`. To add a new type of zone you add it to `JL_TIMING_OWNERS` (and possibly `JL_TIMING_EVENTS`).

From Julia code

The `Compiler.@zone` macro can be used to add a zone from Julia code, it is used as:

```
Compiler.@zone "ZONE NAME" begin
    ...
end
```

Dynamically Enabling and Disabling Zones

The `JULIA_TIMING_SUBSYSTEMS` environment variable allows you to enable or disable zones for a specific Julia run. For instance, setting the variable to `+GC,-INFERENCE` will enable the `GC` zones and disable the `INFERENCE` zones.

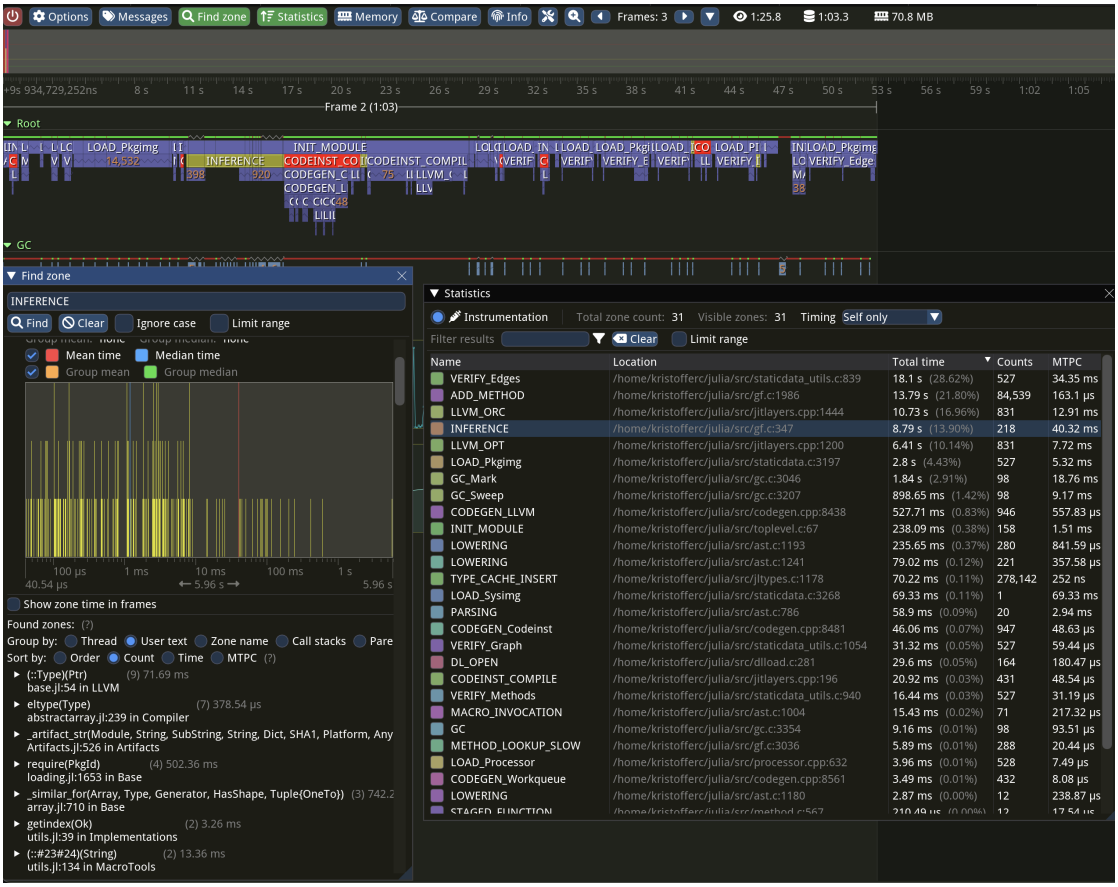


Figure 109.1: Typical Tracy usage

Tracy Profiler

Tracy is a flexible profiler that can be optionally integrated with Julia.

A typical Tracy session might look like this:

Building Julia with Tracy

To enable Tracy integration, build Julia with the extra option `WITH_TRACY=1` in the `Make.user` file.

Installing the Tracy Profile Viewer

The easiest way to obtain the profile viewer is by adding the `TracyProfiler_jll` package and launching the profiler with:

```
run(TracyProfiler_jll.tracy())
```

Note

On macOS, you may want to set the TRACY_DPI_SCALE environment variable to `1.0` if the UI elements in the profiler appear excessively large.

To run a "headless" instance that saves the trace to disk, use

```
run(`$(TracyProfiler_jll.capture()) -o mytracefile.tracy`)
```

instead.

For information on using the Tracy UI, refer to the Tracy manual.

Profiling Julia with Tracy

A typical workflow for profiling Julia with Tracy involves starting Julia using:

```
JULIA_WAIT_FOR_TRACY=1 ./julia -e '...'
```

The environment variable ensures that Julia waits until it has successfully connected to the Tracy profiler before continuing execution. Afterward, use the Tracy profiler UI, click `Connect`, and Julia execution should resume and profiling should start.

Profiling package precompilation with Tracy

To profile a package precompilation process it is easiest to explicitly call into `Base.compilecache` with the package you want to precompile:

```
pkg = Base.identify_package("SparseArrays")
withenv("JULIA_WAIT_FOR_TRACY" => 1, "TRACY_PORT" => 9001) do
    Base.compilecache(pkg)
end
```

Here, we use a custom port for tracy which makes it easier to find the correct client in the Tracy UI to connect to.

Adding metadata to zones

The various `jl_timing_show_*` and `jl_timing_printf` functions can be used to attach a string (or strings) to a zone. For example, the trace zone for inference shows the method instance that is being inferred.

The `TracyCZoneColor` function can be used to set the color of a certain zone. Search through the codebase to see how it is used.

Viewing Tracy files in your browser

Visit https://topolarity.github.io/trace-viewer/ for an (experimental) web viewer for Tracy traces.

You can open a local `.tracy` file or provide a URL from the web (e.g. a file in a Github repo). If you load a trace file from the web, you can also share the page URL directly with others, enabling them to view the same trace.

Enabling stack trace samples

To enable call stack sampling in Tracy, build Julia with these options in your `Make.user` file:

```
WITH_TRACY := 1
WITH_TRACY_CALLSTACKS := 1
USE_BINARYBUILDER_LIBTRACYCLIENT := 0
```

You may also need to run `make -C deps clean-libtracyclient` to force a re-build of Tracy.

This feature has a significant impact on trace size and profiling overhead, so it is recommended to leave call stack sampling off when possible, especially if you intend to share your trace files online.

Note that the Julia JIT runtime does not yet have integration for Tracy's symbolification, so Julia functions will typically be unknown in these stack traces.

Intel VTune (ITTAPI) Profiler

This section is yet to be written.

NVIDIA Nsight Systems (NVTX)

NVTX is NVIDIA's lightweight instrumentation API for annotating events, code ranges, and resources so they can be visualized by NVIDIA profiling tools. Julia emits the same `JL_TIMING` zones used by Tracy and ITTAPI as NVTX ranges, grouped and colored by subsystem, which can then be viewed in NVIDIA Nsight Systems.

NVTX is header-only and its calls are cheap no-ops unless a tool is actively attached and capturing, so building with NVTX support does not require a CUDA toolkit installation and has minimal overhead when no profiler is attached.

Building Julia with NVTX support

To enable NVTX instrumentation of the Julia runtime, build Julia with the extra option `WITH_NVTX := 1` in the `Make.user` file, or with `make ... WITH_NVTX=1` from the command line.

Profiling Julia with Nsight Systems

With Nsight Systems installed, you can record a trace from the command line using `nsys`:

```
nsys profile -o my_julia_trace ./julia -e '...'
```

Open the resulting `my_julia_trace.nsys-rep` file in the Nsight Systems UI to view Julia's timing zones on the NVTX row of the timeline, grouped and colored by subsystem in the same way as with Tracy.

Instrumenting arbitrary Julia code

If you want to additionally annotate your Julia code with custom markers see `NVTX.jl`, which works independently of the `WITH_NVTX=1` build option. You can still use the Nsight Systems profiler as explained above.

109.6 Sanitizer support

Sanitizers can be used in custom Julia builds to make it easier to detect certain kinds of errors in Julia's internal C/C++ code.

Address Sanitizer: easy build

From a source-checkout of Julia, you should be able to build a version supporting address sanitization in Julia and LLVM as follows:

```
$ mkdir /tmp/julia
$ contrib/asan/build.sh /tmp/julia/
```

Here we've chosen `/tmp/julia` as a build directory, but you can choose whatever you wish. Once built, run the workload you wish to test with `/tmp/julia/julia`. Memory bugs will result in errors.

If you require customization or further detail, see the documentation below.

General considerations

Using Clang's sanitizers obviously requires you to use Clang, but there's another catch: most sanitizers require a run-time library, provided by the host compiler, while the instrumented code generated by Julia's JIT relies on functionality from that library. This implies that the LLVM version of your host compiler must match that of the LLVM library used within Julia.

An easy solution is to have a dedicated build folder for providing a matching toolchain, by building with `BUILD_LLVM_CLANG=1`. You can then refer to this toolchain from another build folder by overriding the `CC` and `CXX` variables.

The sanitizers error out when they detect a shared library being opened using `RTLD_DEEPBIND` (ref: google/sanitizers#611). Since libblastrampoline by default uses `RTLD_DEEPBIND`, we need to set the environment variable `LBT_USE_RTLD_DEEPBIND=0` when using a sanitizer.

To use one of the sanitizers set `SANITIZE=1` and then the appropriate flag for the sanitizer you want to use.

On macOS, this might need some extra flags also to work. Altogether, it might look like this, plus one or more of the `SANITIZE_*` flags listed below:

```
make -C deps USE_BINARYBUILDER_LLVM=0 LLVM_VER=svn stage-llvm

make -C src SANITIZE=1 \
    CC=~+/deps/scratch/llvm-svn/build_Release/bin/clang \
    CXX=~+/deps/scratch/llvm-svn/build_Release/bin/clang++ \
    CPPFLAGS="-isysroot $(xcode-select -p)/Platforms/MacOSX.platform/Developer/SDKs/MacOSX.sdk" \
    CXXFLAGS="-isystem $(xcode-select -p)/Toolchains/XcodeDefault.xctoolchain/usr/include/c++/v1"
```

(or put these into your `Make.user`, so you don't need to remember them every time).

Address Sanitizer (ASAN)

For detecting or debugging memory bugs, you can use Clang's address sanitizer (ASAN). By compiling with `SANITIZE_ADDRESS=1` you enable ASAN for the Julia compiler and its generated code. In addition, you can specify `LLVM_SANITIZE=1` to sanitize the LLVM library as well. Note that these options incur a high performance and memory cost. For example, using ASAN for Julia and LLVM makes `testall1` take 8-10 times as long while using 20 times as much memory (this can be reduced to respectively a factor of 3 and 4 by using the options described below).

By default, Julia sets the `allow_user_segv_handler=1` ASAN flag, which is required for signal delivery to work properly. You can define other options using the `ASAN_OPTIONS` environment flag, in which case you'll need to repeat the default option mentioned before. For example, memory usage can be reduced by specifying `fast_unwind_on_malloc=0` and `malloc_context_size=2`, at the cost of backtrace accuracy. For now, Julia also sets `detect_leaks=0`, but this should be removed in the future.

Example setup

Step 1: Install toolchain

Checkout a Git worktree (or create out-of-tree build directory) at `$TOOLCHAIN_WORKTREE` and create a config file `$TOOLCHAIN_WORKTREE/Make.user` with

```
USE_BINARYBUILDER_LLVM=1
BUILD_LLVM_CLANG=1
```

Run:

```
cd $TOOLCHAIN_WORKTREE
make -C deps install-llvm install-clang install-llvm-tools
```

to install toolchain binaries in `$TOOLCHAIN_WORKTREE/usr/tools`

Step 2: Build Julia with ASAN

Checkout a Git worktree (or create out-of-tree build directory) at `$BUILD_WORKTREE` and create a config file `$BUILD_WORKTREE/Make.user` with

```
TOOLCHAIN=$(TOOLCHAIN_WORKTREE)/usr/tools

# use our new toolchain
override CC=$(TOOLCHAIN)/clang
override CXX=$(TOOLCHAIN)/clang++
export ASAN_SYMBOLIZER_PATH=$(TOOLCHAIN)/llvm-symbolizer

USE_BINARYBUILDER_LLVM=1

override SANITIZE=1
override SANITIZE_ADDRESS=1

# make the GC use regular malloc/frees, which are hooked by ASAN
override WITH_GC_DEBUG_ENV=1
```

```
# default to a debug build for better line number reporting
override JULIA_BUILD_MODE=debug

# make ASAN consume less memory
export
↪  ASAN_OPTIONS=detect_leaks=0:fast_unwind_on_malloc=0:allow_user_segv_handler=1:malloc_context_size=2

JULIA_PRECOMPILE=1

# tell libblastrampoline to not use RTLD_DEEPBIND
export LBT_USE_RTLD_DEEPBIND=0
```

Run:

```
cd $BUILD_WORKTREE
make debug
```

to build `julia-debug` with ASAN.

Memory Sanitizer (MSAN)

For detecting use of uninitialized memory, you can use Clang's memory sanitizer (MSAN) by compiling with `SANITIZE_MEMORY=1`.

Thread Sanitizer (TSAN)

For debugging data-races and other threading related issues you can use Clang's thread sanitizer (TSAN) by compiling with `SANITIZE_THREAD=1`.

109.7 Instrumenting Julia with DTrace, and bpftrace

DTrace and bpftrace are tools that enable lightweight instrumentation of processes. You can turn the instrumentation on and off while the process is running, and with instrumentation off the overhead is minimal.

Julia 1.8

Support for probes was added in Julia 1.8

Note

This documentation has been written from a Linux perspective, most of this should hold on Mac OS/Darwin and FreeBSD.

Enabling support

On Linux install the `systemtap` package that has a version of `dtrace` and create a `Make.user` file containing

```
WITH_DTRACE=1
```

to enable USDT probes.

Verifying

```
> readelf -n usr/lib/libjulia-internal.so.1

Displaying notes found in: .note.gnu.build-id
  Owner                Data size  Description
  GNU                  0x00000014 NT_GNU_BUILD_ID (unique build ID bitstring)
    Build ID: 57161002f35548772a87418d2385c284ceb3ead8

Displaying notes found in: .note.stapsdt
  Owner                Data size  Description
  stapsdt              0x00000029 NT_STAPSDT (SystemTap probe descriptors)
    Provider: julia
    Name: gc__begin
    Location: 0x000000000013213e, Base: 0x00000000002bb4da, Semaphore: 0x0000000000346cac
    Arguments:
  stapsdt              0x00000032 NT_STAPSDT (SystemTap probe descriptors)
    Provider: julia
    Name: gc__stop_the_world
    Location: 0x0000000000132144, Base: 0x00000000002bb4da, Semaphore: 0x0000000000346cae
    Arguments:
  stapsdt              0x00000027 NT_STAPSDT (SystemTap probe descriptors)
    Provider: julia
    Name: gc__end
    Location: 0x000000000013214a, Base: 0x00000000002bb4da, Semaphore: 0x0000000000346cb0
    Arguments:
  stapsdt              0x0000002d NT_STAPSDT (SystemTap probe descriptors)
    Provider: julia
    Name: gc__finalizer
    Location: 0x0000000000132150, Base: 0x00000000002bb4da, Semaphore: 0x0000000000346cb2
    Arguments:
```

Adding probes in libjulia

Probes are declared in dtraces format in the file `src/uprobes.d`. The generated header file is included in `src/julia_internal.h` and if you add probes you should provide a noop implementation there.

The header will contain a semaphore `*_ENABLED` and the actual call to the probe. If the probe arguments are expensive to compute you should first check if the probe is enabled and then compute the arguments and call the probe.

```
  if (JL_PROBE_{PROBE}_ENABLED())
    auto expensive_arg = ...;
    JL_PROBE_{PROBE}(expensive_arg);
```

If your probe has no arguments it is preferred to not include the semaphore check. With USDT probes enabled the cost of a semaphore is a memory load, irrespective of the fact that the probe is enabled or not.

```
#define JL_PROBE_GC_BEGIN_ENABLED() __builtin_expect (julia_gc__begin_semaphore, 0)
```

```
__extension__ extern unsigned short julia_gc__begin_semaphore __attribute__ ((unused))
↪  __attribute__ ((section (".probes")));
```

Whereas the probe itself is a noop sled that will be patched to a trampoline to the probe handler.

Available probes

GC probes

1. `julia:gc__begin`: GC begins running on one thread and triggers stop-the-world.
2. `julia:gc__stop_the_world`: All threads have reached a safepoint and GC runs.
3. `julia:gc__mark__begin`: Beginning the mark phase
4. `julia:gc__mark_end(scanned_bytes, perm_scanned)`: Mark phase ended
5. `julia:gc__sweep_begin(full)`: Starting sweep
6. `julia:gc__sweep_end`: Sweep phase finished
7. `julia:gc__end`: GC is finished, other threads continue work
8. `julia:gc__finalizer`: Initial GC thread has finished running finalizers

Task runtime probes

1. `julia:rt__run__task(task)`: Switching to task `task` on current thread.
2. `julia:rt__pause__task(task)`: Switching from task `task` on current thread.
3. `julia:rt__new__task(parent, child)`: Task `parent` created task `child` on current thread.
4. `julia:rt__start__task(task)`: Task `task` started for the first time with a new stack.
5. `julia:rt__finish__task(task)`: Task `task` finished and will no longer execute.
6. `julia:rt__start__process__events(task)`: Task `task` started processing libuv events.
7. `julia:rt__finish__process__events(task)`: Task `task` finished processing libuv events.

Task queue probes

1. `julia:rt__taskq__insert(ptls, task)`: Thread `ptls` attempted to insert `task` into a PARTR multiq.
2. `julia:rt__taskq__get(ptls, task)`: Thread `ptls` popped `task` from a PARTR multiq.

Thread sleep/wake probes

1. `julia:rt__sleep__check__wake(ptls, old_state)`: Thread (PTLS `ptls`) waking up, previously in state `old_state`.
2. `julia:rt__sleep__check__wakeup(ptls)`: Thread (PTLS `ptls`) woke itself up.
3. `julia:rt__sleep__check__sleep(ptls)`: Thread (PTLS `ptls`) is attempting to sleep.
4. `julia:rt__sleep__check__taskq__wake(ptls)`: Thread (PTLS `ptls`) fails to sleep due to tasks in PARTR multiq.
5. `julia:rt__sleep__check__task__wake(ptls)`: Thread (PTLS `ptls`) fails to sleep due to tasks in Base workqueue.
6. `julia:rt__sleep__check__uv__wake(ptls)`: Thread (PTLS `ptls`) fails to sleep due to libuv wakeup.

Probe usage examples

GC stop-the-world latency

An example `bpftrace` script is given in `contrib/gc_stop_the_world_latency.bt` and it creates a histogram of the latency for all threads to reach a safepoint.

Running this Julia code, with `julia -t 2`

```
using Base.Threads

fib(x) = x <= 1 ? 1 : fib(x-1) + fib(x-2)

beaver = @spawn begin
    while true
        fib(30)
        # A manual safepoint is necessary since otherwise this loop
        # may never yield to GC.
        GC.safepoint()
    end
end

allocator = @spawn begin
    while true
        zeros(1024)
    end
end

wait(allocator)
```

and in a second terminal

```
> sudo contrib/bpftrace/gc_stop_the_world_latency.bt
Attaching 4 probes...
Tracing Julia GC Stop-The-World Latency... Hit Ctrl-C to end.
^C
```

```
@usecs[1743412]:
[4, 8)               971 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@|
[8, 16)              837 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@        |
[16, 32)             129 |@@@@@@                                              |
[32, 64)              10 |                                                    |
[64, 128)              1 |                                                    |
```

We can see the latency distribution of the stop-the-world phase in the executed Julia process.

Task spawn monitor

It's sometimes useful to know when a task is spawning other tasks. This is very easy to see with `rt__new__task`. The first argument to the probe, `parent`, is the existing task which is creating a new task. This means that if you know the address of the task you want to monitor, you can easily just look at the tasks that were spawned by that specific task. Let's see how to do this; first let's start a Julia session and get the PID and REPL's task address:

```
> julia
               _
   _       _ _(_)_     |  Documentation: https://docs.julialang.org
  (_)     | (_) (_)    |
   _ _   _| |_  __ _   |  Type "?" for help, "]?" for Pkg help.
  | | | | | | |/ _` |  |
  | | |_| | | | (_| |  |  Version 1.6.2 (2021-07-14)
 _/ |\__'_|_|_|\__'_|  |  Official https://julialang.org release
|__/                   |

1> getpid()
997825

2> current_task()
Task (runnable) @0x00007f524d088010
```

Now we can start `bpftrace` and have it monitor `rt__new__task` for *only* this parent:

`sudo bpftrace -p 997825 -e 'usdt:usr/lib/libjulia-internal.so:julia:rt__new__task /arg0==0x00007f524d088010/{ printf("Task: %x\n", arg0); }'`

(Note that in the above, `arg0` is the first argument, `parent`).

And if we spawn a single task:

`Threads.@spawn 1+1`

we see this task being created:

`Task: 4d088010`

However, if we spawn a bunch of tasks from that newly-spawned task:

```julia
Threads.@spawn for i in 1:10
   Threads.@spawn 1+1
end
```

we still only see one task from `bpftrace`:

```
Task: 4d088010
```

and it's still the same task we were monitoring! Of course, we can remove this filter to see *all* newly-created tasks just as easily:

```
sudo bpftrace -p 997825 -e 'usdt:usr/lib/libjulia-internal.so:julia:rt__new__task { printf("Task: %x\n", arg0); }'
```

```
Task: 4d088010
Task: 4dc4e290
Task: 4dc4e290
Task: 4dc4e290
Task: 4dc4e290
Task: 4dc4e290
Task: 4dc4e290
Task: 4dc4e290
Task: 4dc4e290
Task: 4dc4e290
Task: 4dc4e290
```

We can see our root task, and the newly-spawned task as the parent of the ten even newer tasks.

Thundering herd detection

Task runtimes can often suffer from the "thundering herd" problem: when some work is added to a quiet task runtime, all threads may be woken up from their slumber, even if there isn't enough work for each thread to process. This can cause extra latency and CPU cycles while all threads awaken (and simultaneously go back to sleep, not finding any work to execute).

We can see this problem illustrated with `bpftrace` quite easily. First, in one terminal we start Julia with multiple threads (6 in this example), and get the PID of that process:

```
> julia -t 6
               _
   _       _ _(_)_     |  Documentation: https://docs.julialang.org
  (_)     | (_) (_)    |
   _ _   _| |_  __ _   |  Type "?" for help, "]?" for Pkg help.
  | | | | | | |/ _` |  |
  | | |_| | | | (_| |  |  Version 1.6.2 (2021-07-14)
 _/ |\__'_|_|_|\__'_|  |  Official https://julialang.org release
|__/                   |

1> getpid()
997825
```

And in another terminal we start `bpftrace` monitoring our process, specifically probing the `rt__sleep__check__wake` hook:

```
sudo bpftrace -p 997825 -e 'usdt:usr/lib/libjulia-internal.so:julia:rt__sleep__check__wake { printf("Thread wake up! %x\n", arg0); }'
```

Now, we create and execute a single task in Julia:

`Threads.@spawn 1+1`

And in `bpftrace` we see printed out something like:

```
Thread wake up! 3f926100
Thread wake up! 3ebd5140
Thread wake up! 3f876130
Thread wake up! 3e2711a0
Thread wake up! 3e312190
```

Even though we only spawned a single task (which only one thread could process at a time), we woke up all of our other threads! In the future, a smarter task runtime might only wake up a single thread (or none at all; the spawning thread could execute this task!), and we should see this behavior go away.

Task Monitor with BPFnative.jl

BPFnative.jl is able to attach to USDT probe points just like `bpftrace`. There is a demo available for monitoring the task runtime, GC, and thread sleep/wake transitions here.

Notes on using `bpftrace`

An example probe in the bpftrace format looks like:

```
usdt:usr/lib/libjulia-internal.so:julia:gc__begin
{
  @start[pid] = nsecs;
}
```

The probe declaration takes the kind `usdt`, then either the path to the library or the PID, the provider name `julia` and the probe name `gc__begin`. Note that I am using a relative path to the `libjulia-internal.so`, but this might need to be an absolute path on a production system.

Useful references:

- Julia Evans blog on Linux tracing systems
- LWN article on USDT and BPF
- GDB support for probes
- Brendan Gregg – Linux Performance

Chapter 110

Building Julia

110.1 Building Julia (Detailed)

Downloading the Julia source code

If you are behind a firewall, you may need to use the `https` protocol instead of the `git` protocol:

```
git config --global url."https://".insteadOf git://
```

Be sure to also configure your system to use the appropriate proxy settings, e.g. by setting the `https_proxy` and `http_proxy` variables.

Building Julia

When compiled the first time, the build will automatically download pre-built external dependencies. If you prefer to build all the dependencies on your own, or are building on a system that cannot access the network during the build process, add the following in `Make.user`:

```
USE_BINARYBUILDER=0
```

Building Julia requires 5GiB if building all dependencies and approximately 4GiB of virtual memory.

To perform a parallel build, use `make -j N` and supply the maximum number of concurrent processes. If the defaults in the build do not work for you, and you need to set specific make parameters, you can save them in `Make.user`, and place the file in the root of your Julia source. The build will automatically check for the existence of `Make.user` and use it if it exists.

You can create out-of-tree builds of Julia by specifying `make O=<build-directory> configure` on the command line. This will create a directory mirror, with all of the necessary Makefiles to build Julia, in the specified directory. These builds will share the source files in Julia and `deps/srccache`. Each out-of-tree build directory can have its own `Make.user` file to override the global `Make.user` file in the top-level folder. After modifying the `Make.user` file if necessary, build using: `make -C <build-directory>`.

If everything works correctly, there will be a symlink to the julia executable in the build directory which can be run as:

```
./julia
```

The actual executable is in <build-directory>/usr/bin. After running this, you will see a Julia banner and an interactive prompt into which you can enter expressions for evaluation. (Errors related to libraries might be caused by old, incompatible libraries sitting around in your PATH. In this case, try moving the julia directory earlier in the PATH). Note that most of the instructions above apply to unix systems.

To run julia from anywhere you can:

- add an alias (in bash: echo "alias julia='<build-directory>/usr/bin/julia'" >> ~/.bashrc && source ~/.bashrc), or
- add a soft link to the julia executable in the <build-directory>/usr/bin directory to /usr/local/bin (or any suitable directory already in your path), or
- add the julia directory to your executable path for this shell session (in bash: export PATH="$(pwd):$PATH" ; in csh or tcsh:

set path= ($path $cwd)), or

- add the julia directory to your executable path permanently (e.g. in .bash_profile), or
- write prefix=/path/to/install/folder into Make.user and then run make install. If there is a version of Julia already installed in this folder, you should delete it before running make install.

Some of the options you can set to control the build of Julia are listed and documented at the beginning of the file Make.inc, but you should never edit it for this purpose, use Make.user instead.

Julia's Makefiles define convenient automatic rules called print-<VARNAME> for printing the value of variables, replacing <VARNAME> with the name of the variable to print the value of. For example

```
$ make print-JULIA_PRECOMPILE
JULIA_PRECOMPILE=1
```

These rules are useful for debugging purposes.

Now you should be able to run Julia like this:

```
julia
```

If you are building a Julia package for distribution on Linux, macOS, or Windows, take a look at the detailed notes in distributing.md.

Updating an existing source tree

If you have previously downloaded julia using git clone, you can update the existing source tree using git pull rather than starting anew:

```
cd julia
git pull && make
```

Assuming that you had made no changes to the source tree that will conflict with upstream updates, these commands will trigger a build to update to the latest version.

General troubleshooting

1. Over time, the base library may accumulate enough changes such that the bootstrapping process in building the system image will fail. If this happens, the build may fail with an error like

   ```
   *** This error is usually fixed by running 'make clean'. If the error persists, try 'make
   ↪  cleanall' ***
   ```

   As described, running `make clean && make` is usually sufficient. Occasionally, the stronger cleanup done by `make cleanall` is needed.

2. New versions of external dependencies may be introduced which may occasionally cause conflicts with existing builds of older versions.

   a. Special make targets exist to help wipe the existing build of a dependency. For example, `make -C deps clean-llvm` will clean out the existing build of `llvm` so that `llvm` will be rebuilt from the downloaded source distribution the next time `make` is called. `make -C deps distclean-llvm` is a stronger wipe which will also delete the downloaded source distribution, ensuring that a fresh copy of the source distribution will be downloaded and that any new patches will be applied the next time `make` is called.

   b. To delete existing binaries of `julia` and all its dependencies, delete the `./usr` directory *in the source tree*.

3. If you've updated macOS recently, be sure to run `xcode-select --install` to update the command line tools. Otherwise, you could run into errors for missing headers and libraries, such as `ld: library not found for -lcrt1.10.6.o`.

4. If you've moved the source directory, you might get errors such as `CMake Error: The current CMakeCache.txt directory ... is different than the directory ... where CMakeCache.txt was created.`, in which case you may delete the offending dependency under deps

5. In extreme cases, you may wish to reset the source tree to a pristine state. The following git commands may be helpful:

   ```
   git reset --hard #Forcibly remove any changes to any files under version control
   git clean -x -f -d #Forcibly remove any file or directory not under version control
   ```

   To avoid losing work, make sure you know what these commands do before you run them. `git` will not be able to undo these changes!

Platform-Specific Notes

Notes for various operating systems:

- Linux
- macOS
- Windows
- FreeBSD

Notes for various architectures:

- ARM
- RISC-V

Required Build Tools and External Libraries

Building Julia requires that the following software be installed:

- **GNU make** — building dependencies.
- **gcc & g++** (>= 7.1) or **Clang** (>= 5.0, >= 9.3 for Apple Clang) — compiling and linking C, C++.
- **libatomic** — provided by **[gcc]** and needed to support atomic operations.
- **python** (>=2.7) — needed to build LLVM.
- **gfortran** — compiling and linking Fortran libraries.
- **perl** — preprocessing of header files of libraries.
- **wget**, **curl**, or **fetch** (FreeBSD) — to automatically download external libraries.
- **m4** — needed to build GMP.
- **awk** — helper tool for Makefiles.
- **patch** — for modifying source code.
- **cmake** (>= 3.4.3) — needed to build `libgit2`.
- **pkg-config** — needed to build `libgit2` correctly, especially for proxy support.
- **powershell** (>= 3.0) — necessary only on Windows.
- **which** — needed for checking build dependencies.

On Debian-based distributions (e.g. Ubuntu), you can easily install them with `apt-get`:

```
sudo apt-get install build-essential libatomic1 python gfortran perl wget m4 cmake pkg-config
↪  curl
```

On Red Hat-based distributions (e.g. Fedora, CentOS), you can install them with `yum`:

```
sudo dnf install gcc gcc-c++ gcc-gfortran python3 perl wget m4 cmake pkgconfig curl
```

Julia uses the following external libraries, which are automatically downloaded (or in a few cases, included in the Julia source repository) and then compiled from source the first time you run `make`. The specific version numbers of these libraries that Julia uses are listed in `deps/$(libname).version`:

- **LLVM** (15.0 + patches) — compiler infrastructure (see note below).
- **FemtoLisp** — packaged with Julia source, and used to implement the compiler front-end.

- **libuv** (custom fork) — portable, high-performance event-based I/O library.
- **OpenLibm** — portable libm library containing elementary math functions.
- **DSFMT** — fast Mersenne Twister pseudorandom number generator library.
- **OpenBLAS** — fast, open, and maintained [basic linear algebra subprograms (BLAS)]
- **LAPACK** — library of linear algebra routines for solving systems of simultaneous linear equations, least-squares solutions of linear systems of equations, eigenvalue problems, and singular value problems.
- **MKL** (optional) – OpenBLAS and LAPACK may be replaced by Intel's MKL library.
- **SuiteSparse** — library of linear algebra routines for sparse matrices.
- **PCRE** — Perl-compatible regular expressions library.
- **GMP** — GNU multiple precision arithmetic library, needed for `BigInt` support.
- **MPFR** — GNU multiple precision floating point library, needed for arbitrary precision floating point (`BigFloat`) support.
- **libgit2** — Git linkable library, used by Julia's package manager.
- **curl** — libcurl provides download and proxy support.
- **libssh2** — library for SSH transport, used by libgit2 for packages with SSH remotes.
- **OpenSSL** — library used for cryptography and transport layer security, used by libgit2 and libssh2.
- **utf8proc** — a library for processing UTF-8 encoded Unicode strings.
- **LLVM libunwind** — LLVM's fork of [libunwind], a library that determines the call-chain of a program.
- **ITTAPI** — Intel's Instrumentation and Tracing Technology and Just-In-Time API.

Build dependencies

If you already have one or more of these packages installed on your system, you can prevent Julia from compiling duplicates of these libraries by passing `USE_SYSTEM_...=1` to `make` or adding the line to `Make.user`. The complete list of possible flags can be found in `Make.inc`.

Please be aware that this procedure is not officially supported, as it introduces additional variability into the installation and versioning of the dependencies, and is recommended only for system package maintainers. Unexpected compile errors may result, as the build system will do no further checking to ensure the proper packages are installed.

LLVM

The most complicated dependency is LLVM, for which we require additional patches from upstream (LLVM is not backward compatible).

For packaging Julia with LLVM, we recommend either:

- bundling a Julia-only LLVM library inside the Julia package, or

- adding the patches to the LLVM package of the distribution.
  - A complete list of patches is available in on Github see the `julia-release/18.x` branch.
  - The remaining patches are all upstream bug fixes, and have been contributed into upstream LLVM.

Using an unpatched or different version of LLVM will result in errors and/or poor performance. You can build a different version of LLVM from a remote Git repository with the following options in the `Make.user` file:

```
# Force source build of LLVM
USE_BINARYBUILDER_LLVM = 0
# Use Git for fetching LLVM source code
# this is either `1` to get all of them
DEPS_GIT = 1
# or a space-separated list of specific dependencies to download with git
DEPS_GIT = llvm

# Other useful options:
#URL of the Git repository you want to obtain LLVM from:
#  LLVM_GIT_URL = ...
#Name of the alternate branch to clone from git
#  LLVM_BRANCH = julia-16.0.6-0
#SHA hash of the alternate commit to check out automatically
#  LLVM_SHA1 = $(LLVM_BRANCH)
#List of LLVM targets to build. It is strongly recommended to keep at least all the
#default targets listed in `deps/llvm.mk`, even if you don't necessarily need all of them.
#  LLVM_TARGETS = ...
#Use ccache for faster recompilation in case you need to restart a build.
#  USECCACHE = 1
#  CMAKE_GENERATOR=Ninja
#  LLVM_ASSERTIONS=1
#  LLVM_DEBUG=Symbols
```

The various build phases are controlled by specific files:

- `deps/llvm.version` : touch or change to checkout a new version, `make get-llvm check-llvm`
- `deps/srccache/llvm/source-extracted` : result of `make extract-llvm`
- `deps/llvm/build_Release*/build-configured` : result of `make configure-llvm`
- `deps/llvm/build_Release*/build-configured` : result of `make compile-llvm`
- `usr-staging/llvm/build_Release*.tgz` : result of `make stage-llvm` (regenerate with `make reinstall-llvm`)
- `usr/manifest/llvm` : result of `make install-llvm` (regenerate with `make uninstall-llvm`)
- `make version-check-llvm` : runs every time to warn the user if there are local modifications

Though Julia can be built with newer LLVM versions, support for this should be regarded as experimental and not suitable for packaging.

libuv

Julia uses a custom fork of libuv. It is a small dependency, and can be safely bundled in the same package as Julia, and will not conflict with the system library. Julia builds should *not* try to use the system libuv.

BLAS and LAPACK

As a high-performance numerical language, Julia should be linked to a multi-threaded BLAS and LAPACK, such as OpenBLAS or ATLAS, which will provide much better performance than the reference `libblas` implementations which may be default on some systems.

Source distributions of releases

Each pre-release and release of Julia has a "full" source distribution and a "light" source distribution.

The full source distribution contains the source code for Julia and all dependencies so that it can be built from source without an internet connection. The light source distribution does not include the source code of dependencies.

For example, `julia-1.0.0.tar.gz` is the light source distribution for the `v1.0.0` release of Julia, while `julia-1.0.0-full.tar.gz` is the full source distribution.

Building Julia from source with a Git checkout of a stdlib

If you need to build Julia from source with a Git checkout of a stdlib, then use `make DEPS_GIT=NAME_OF_STDLIB` when building Julia.

For example, if you need to build Julia from source with a Git checkout of Pkg, then use `make DEPS_GIT=Pkg` when building Julia. The Pkg repo is in `stdlib/Pkg`, and created initially with a detached `HEAD`. If you're doing this from a pre-existing Julia repository, you may need to `make clean` beforehand.

If you need to build Julia from source with Git checkouts of more than one stdlib, then `DEPS_GIT` should be a space-separated list of the stdlib names. For example, if you need to build Julia from source with a Git checkout of Pkg, Tar, and Downloads, then use `make DEPS_GIT='Pkg Tar Downloads'` when building Julia.

Building an "assert build" of Julia

An "assert build" of Julia is a build that was built with both `FORCE_ASSERTIONS=1` and `LLVM_ASSERTIONS=1`. To build an assert build, define both of the following variables in your `Make.user` file:

```
FORCE_ASSERTIONS=1
LLVM_ASSERTIONS=1
```

Please note that assert builds of Julia will be slower than regular (non-assert) builds.

Building a debug build of Julia

A full debug build of Julia can be built with `make debug`. This builds a debug version of `libjulia` and uses it to bootstrap the compiler, before creating a system image with debug symbols enabled. This can take more than 15 minutes.

Although it may result in some differences, a debug build can be built much quicker by bootstrapping from a release build:

```
$ make julia-src-release julia-sysbase-release
$ make julia-sysimg-debug CROSS_BOOTSTRAP_JULIA=$PWD/usr/bin/julia
↪  CROSS_BOOTSTRAP_SYSBASE=$PWD/usr/lib/julia/sysbase.so
```

Building 32-bit Julia on a 64-bit machine

Occasionally, bugs specific to 32-bit architectures may arise, and when this happens it is useful to be able to debug the problem on your local machine. Since most modern 64-bit systems support running programs built for 32-bit ones, if you don't have to recompile Julia from source (e.g. you mainly need to inspect the behavior of a 32-bit Julia without having to touch the C code), you can likely use a 32-bit build of Julia for your system that you can obtain from the official downloads page. However, if you do need to recompile Julia from source one option is to use a Docker container of a 32-bit system. At least for now, building a 32-bit version of Julia is relatively straightforward using ubuntu 32-bit docker images. In brief, after setting up `docker` here are the required steps:

```
$ docker pull i386/ubuntu
$ docker run --platform i386 -i -t i386/ubuntu /bin/bash
```

At this point you should be in a 32-bit machine console (note that `uname` reports the host architecture, so will still say 64-bit, but this will not affect the Julia build). You can add packages and compile code; when you `exit`, all the changes will be lost, so be sure to finish your analysis in a single session or set up a copy/pastable script you can use to set up your environment.

From this point, you should

```
# apt update
```

(Note that `sudo` isn't installed, but neither is it necessary since you are running as `root`, so you can omit `sudo` from all commands.)

Then add all the build dependencies, a console-based editor of your choice, `git`, and anything else you'll need (e.g., `gdb`, `rr`, etc). Pick a directory to work in and `git clone` Julia, check out the branch you wish to debug, and build Julia as usual.

Update the version number of a dependency

There are two types of builds

1. Build everything (`deps/` and `src/`) from source code. (Add `USE_BINARYBUILDER=0` to `Make.user`, see Building Julia)
2. Build from source (`src/`) with pre-compiled dependencies (default)

When you want to update the version number of a dependency in `deps/`, you may want to use the following checklist:

```
### Check list

Version numbers:
```

```
- [ ] `deps/$(libname).version`: `LIBNAME_VER`, `LIBNAME_BRANCH`, `LIBNAME_SHA1` and
↪  `LIBNAME_JLL_VER`
- [ ] `stdlib/$(LIBNAME_JLL_NAME)_jll/Project.toml`: `version`

Checksum:
- [ ] `deps/checksums/$(libname)`
- [ ] `deps/checksums/$(LIBNAME_JLL_NAME)-*/`: `md5` and `sha512`

Patches:
- [ ] `deps/$(libname).mk`
- [ ] `deps/patches/$(libname)-*.patch`
```

Note:

- For specific dependencies, some items in the checklist may not exist.
- For checksum file, it may be **a single file** without a suffix, or **a folder** containing two files.

Example: `OpenLibm`

1. Update Version numbers in `deps/openlibm.version`
   - `OPENLIBM_VER := 0.X.Y`
   - `OPENLIBM_BRANCH = v0.X.Y`
   - `OPENLIBM_SHA1 = new-sha1-hash`
2. Update Version number in `stdlib/OpenLibm_jll/Project.toml`
   - `version = "0.X.Y+0"`
3. Update checksums in `deps/checksums/openlibm`
   - `make -f contrib/refresh_checksums.mk openlibm`
4. Check if the patch files `deps/patches/openlibm-*.patch` exist
   - if patches don't exist, skip.
   - if patches exist, check if they have been merged into the new version and need to be removed. When deleting a patch, remember to modify the corresponding Makefile file (`deps/openlibm.mk`).

110.2 Linux

- GCC version 4.7 or later is required to build Julia.
- To use external shared libraries not in the system library search path, set `USE_SYSTEM_XXX=1` and `LDFLAGS=-Wl,-rpath,/path/to/dir/contains/libXXX.so` in `Make.user`.
- Instead of setting LDFLAGS, putting the library directory into the environment variable `LD_LIBRARY_PATH` (at both compile and run time) also works.

- The USE_SYSTEM_* flags should be used with caution. These are meant only for troubleshooting, porting, and packaging, where package maintainers work closely with the Julia developers to make sure that Julia is built correctly. Production use cases should use the officially provided binaries. Issues arising from the use of these flags will generally not be accepted.
- See also the external dependencies.

Architecture Customization

Julia can be built for a non-generic architecture by configuring the ARCH Makefile variable in a Make.user file. See the appropriate section of Make.inc for additional customization options, such as MARCH and JULIA_CPU_TARGET.

For example, to build for Pentium 4, set MARCH=pentium4 and install the necessary system libraries for linking. On Ubuntu, these may include lib32gfortran-6-dev, lib32gcc1, and lib32stdc++6, among others.

You can also set MARCH=native in Make.user for a maximum-performance build customized for the current machine CPU.

Linux Build Troubleshooting

Problem	Possible Solution
OpenBLAS build failure	Set one of the following build options in Make.user and build again: - OPENBLAS_TARGET_ARCH=BARCELONA (AMD CPUs) or OPENBLAS_TARGET_ARCH=NEHALEM (Intel CPUs) Set OPENBLAS_DYNAMIC_ARCH = 0 to disable compiling multiple architectures in a single binary. - OPENBLAS_NO_AVX2 = 1 disables AVX2 instructions, allowing OpenBLAS to compile with OPENBLAS_DYNAMIC_ARCH = 1 using old versions of binutils - USE_SYSTEM_BLAS=1 uses the system provided libblas - Set LIBBLAS=-lopenblas and LIBBLASNAME=libopenblas to force the use of the system provided OpenBLAS when multiple BLAS versions are installed. If you get an error that looks like ../kernel/x86_64/dgemm_kernel_4x4_haswell.S:1709: Error: no such instruction: `vpermpd $ 0xb1,%ymm0,%ymm0', then you need to set OPENBLAS_DYNAMIC_ARCH = 0 or OPENBLAS_NO_AVX2 = 1, or you need a newer version of binutils (2.18 or newer). (Issue #7653) If the linker cannot find gfortran and you get an error like julia /usr/bin/x86_64-linux-gnu-ld: cannot find -lgfortran, check the path with gfortran -print-file-name=libgfortran.so and use the output to export something similar to this: export LDFLAGS=-L/usr/lib/gcc/x86_64-linux-gnu/8/. See Issue #6150.
Illegal Instruction error	Check if your CPU supports AVX while your OS does not (e.g. through virtualization, as described in this issue).

110.3 macOS

You need to have the current Xcode command line utilities installed: run xcode-select --install in the terminal. You will need to rerun this terminal command after each macOS update, otherwise you may run into errors involving missing libraries or headers.

The dependent libraries are now built with BinaryBuilder and will be automatically downloaded. This is the preferred way to build Julia source. In case you want to build them all on your own, you will need a 64-bit gfortran to compile Julia dependencies.

```
brew install gcc
```

If you have set `LD_LIBRARY_PATH` or `DYLD_LIBRARY_PATH` in your `.bashrc` or equivalent, Julia may be unable to find various libraries that come bundled with it. These environment variables need to be unset for Julia to work.

110.4 Windows

This file describes how to install, or build, and use Julia on Windows.

For more general information about Julia, please see the main README or the documentation.

General Information for Windows

We highly recommend running Julia using a modern terminal application, in particular Windows Terminal, which can be installed from the Microsoft Store.

Line endings

Julia uses binary-mode files exclusively. Unlike many other Windows programs, if you write `\n` to a file, you get a `\n` in the file, not some other bit pattern. This matches the behavior exhibited by other operating systems. If you have installed Git for Windows, it is suggested, but not required, that you configure your system Git to use the same convention:

```
git config --global core.eol lf
git config --global core.autocrlf input
```

or edit `%USERPROFILE%\.gitconfig` and add/edit the lines:

```
[core]
    eol = lf
    autocrlf = input
```

Binary distribution

For the binary distribution installation notes on Windows please see the instructions at https://julialang.org/downloads/platform/#windows. Note, however, that on all platforms using `juliaup` is recommended over manually installing binaries.

Source distribution

Cygwin-to-MinGW cross-compiling

The recommended way of compiling Julia from source on Windows is by cross compiling from Cygwin, using versions of the MinGW-w64 compilers available through Cygwin's package manager.

1. Download and run Cygwin setup for 32 bit or 64 bit. Note, that you can compile either 32 or 64 bit Julia from either 32 or 64 bit Cygwin. 64 bit Cygwin has a slightly smaller but often more up-to-date selection of packages.

   Advanced: you may skip steps 2-4 by running:

   ```
   setup-x86_64.exe -s <url> -q -P
   ↪  cmake,gcc-g++=12.5.0-1,git,make,patch,curl,m4,python3,p7zip,mingw64-i686-gcc-g++=12.5.0-1,mingw64-i686-gcc-fo
   ```

   replacing `<url>` with a site from https://cygwin.com/mirrors.html or run setup manually first and select a mirror.

2. Select installation location and a mirror to download from.

3. At the *Select Packages* step, select the following:

   1. From the *Devel* category: `cmake`, `gcc-g++`, `git`, `make`, `patch`
   2. From the *Net* category: `curl`
   3. From *Interpreters* (or *Python*) category: `m4`, `python3`
   4. From the *Archive* category: `p7zip`
   5. For 32 bit Julia, and also from the *Devel* category: `mingw64-i686-gcc-g++` and `mingw64-i686-gcc-fortran` and `mingw64-i686-gcc-core` (version "12.5.0-1") `mingw64-i686-headers` and `mingw64-i686-runtime` and `mingw64-i686-winpthreads` (version "12.0.0-1")
   6. For 64 bit Julia, and also from the *Devel* category: `mingw64-x86_64-gcc-g++` and `mingw64-x86_64-gcc-fortran` and `mingw64-x86_64-gcc-core` (version "12.5.0-1") `mingw64-x86_64-headers` and `mingw64-x86_64-runtime` and `mingw64-x86_64-winpthreads` (version "12.0.0-1")

4. Allow Cygwin installation to finish, then start from the installed shortcut '*Cygwin Terminal*', or '*Cygwin64 Terminal*', respectively.

5. Build Julia and its dependencies from source:

   1. Get the Julia sources

      ```
      git clone https://github.com/JuliaLang/julia.git
      cd julia
      ```

      Tip: If you get an `error: cannot fork() for fetch-pack: Resource temporarily unavailable` from git, add `alias git="env PATH=/usr/bin git"` to `~/.bashrc` and restart Cygwin.

   2. Set the `XC_HOST` variable in `Make.user` to indicate MinGW-w64 cross compilation

      ```
      echo 'XC_HOST = i686-w64-mingw32' > Make.user     # for 32 bit Julia
      # or
      echo 'XC_HOST = x86_64-w64-mingw32' > Make.user   # for 64 bit Julia
      ```

   3. Start the build

      ```
      make -j 4       # Adjust the number of threads (4) to match your build environment.
      make -j 4 debug # This builds julia-debug.exe
      ```

6. Run Julia using the Julia executables directly

   ```
   usr/bin/julia.exe
   usr/bin/julia-debug.exe
   ```

Pro tip: build both!

```
make O=julia-win32 configure
make O=julia-win64 configure
echo 'XC_HOST = i686-w64-mingw32' > julia-win32/Make.user
echo 'XC_HOST = x86_64-w64-mingw32' > julia-win64/Make.user
echo 'ifeq ($(BUILDROOT),$(JULIAHOME))
        $(error "in-tree build disabled")
      endif' >> Make.user
make -C julia-win32  # build for Windows x86 in julia-win32 folder
make -C julia-win64  # build for Windows x86-64 in julia-win64 folder
```

Compiling with MinGW/MSYS2

MSYS2 is a software distribution and build environment for Windows.

Note: MSYS2 requires **64 bit** Windows 7 or newer.

1. Install and configure MSYS2.
    1. Download and run the latest installer for the 64-bit distribution. The installer will have a name like `msys2-x86_64-yyyymmdd.exe`.
    2. Open the MSYS2 shell. Update the package database and base packages:
        ```
        pacman -Syu
        ```
    3. Exit and restart MSYS2. Update the rest of the base packages:
        ```
        pacman -Syu
        ```
    4. Then install tools required to build julia:
        ```
        pacman -S diffutils git m4 make patch tar p7zip curl python
        ```
        For 64 bit Julia, install the x86_64 version:
        ```
        pacman -S mingw-w64-x86_64-gcc mingw-w64-x86_64-cmake mingw-w64-x86_64-clang
        ```
        For 32 bit Julia, install the i686 version:
        ```
        pacman -S mingw-w64-i686-gcc mingw-w64-i686-cmake mingw-w64-i686-clang
        ```
    5. Configuration of MSYS2 is complete. Now `exit` the MSYS2 shell.
2. Build Julia and its dependencies with pre-build dependencies.
    1. Open a new **MINGW64/MINGW32 shell**. Currently we can't use both mingw32 and mingw64, so if you want to build the x86_64 and i686 versions, you'll need to build them in each environment separately.
    2. Clone the Julia sources:
        ```
        git clone https://github.com/JuliaLang/julia.git
        cd julia
        ```
    3. If you want to use clang (currently required if building LLVM from source), put the following in your Make.user

```
CC=/mingw64/bin/clang CXX=/mingw64/bin/clang++
```

UCRT Unsupported

Do not try to use any other clang that MSYS2 may install (which may not have the correct default target) or the "Clang" environment(which defaults to the currently unsupported ucrt).

```
4. Start the build

   ```
   make -j$(nproc)
   ```
```

Pro tip: build in dir

```
make O=julia-mingw-w64 configure
echo 'ifeq ($(BUILDROOT),$(JULIAHOME))
        $(error "in-tree build disabled")
      endif' >> Make.user
make -C julia-mingw-w64
```

Cross-compiling from Unix (Linux/Mac/WSL)

You can also use MinGW-w64 cross compilers to build a Windows version of Julia from Linux, Mac, or the Windows Subsystem for Linux (WSL).

First, you will need to ensure your system has the required dependencies. We need wine (>=1.7.5), a system compiler, and some downloaders. Note: a Cygwin install might interfere with this method if using WSL.

On Ubuntu (on other Linux systems the dependency names are likely to be similar):

```
apt-get install wine-stable gcc wget p7zip-full winbind mingw-w64 gfortran-mingw-w64
dpkg --add-architecture i386 && apt-get update && apt-get install wine32 # add sudo to each if
↪  needed
# switch all of the following to their "-posix" variants (interactively):
for pkg in i686-w64-mingw32-g++ i686-w64-mingw32-gcc i686-w64-mingw32-gfortran
↪  x86_64-w64-mingw32-g++ x86_64-w64-mingw32-gcc x86_64-w64-mingw32-gfortran; do
    sudo update-alternatives --config $pkg
done
```

On Mac: Install XCode, XCode command line tools, X11 (now XQuartz), and MacPorts or Homebrew. Then run `port install wine wget mingw-w64`, or `brew install wine wget mingw-w64`, as appropriate.

Then run the build:

1. `git clone https://github.com/JuliaLang/julia.git julia-win32`
2. `cd julia-win32`
3. `echo override XC_HOST = i686-w64-mingw32 >> Make.user`

4. `make`
5. `make win-extras` (Necessary before running `make binary-dist`)
6. `make binary-dist` then `make exe` to create the Windows installer.
7. move the `julia-*.exe` installer to the target machine

If you are building for 64-bit Windows, the steps are essentially the same. Just replace `i686` in `XC_HOST` with `x86_64`. (Note: on Mac, wine only runs in 32-bit mode).

Debugging a cross-compiled build under wine

The most effective way to debug a cross-compiled version of Julia on the cross-compilation host is to install a Windows version of GDB and run it under wine as usual. The pre-built packages available as part of the MSYS2 project are known to work. Apart from the GDB package you may also need the python and termcap packages. Finally, GDB's prompt may not work when launched from the command line. This can be worked around by prepending `wineconsole` to the regular GDB invocation.

After compiling

Compiling using one of the options above creates a basic Julia build, but not some extra components that are included if you run the full Julia binary installer. If you need these components, the easiest way to get them is to build the installer yourself using `make win-extras` followed by `make binary-dist` and `make exe`. Then run the resulting installer.

Windows Build Debugging

GDB hangs with Cygwin mintty

- Run GDB under the Windows console (cmd) instead. GDB may not function properly under mintty with non- Cygwin applications. You can use `cmd /c start` to start the Windows console from mintty if necessary.

GDB not attaching to the right process

- Use the PID from the Windows task manager or `WINPID` from the `ps` command instead of the PID from unix-style command line tools (e.g. `pgrep`). You may need to add the PID column if it is not shown by default in the Windows task manager.

GDB not showing the right backtrace

- When attaching to the julia process, GDB may not be attaching to the right thread. Use `info threads` command to show all the threads and `thread <threadno>` to switch threads.
- Be sure to use a 32 bit version of GDB to debug a 32 bit build of Julia, or a 64 bit version of GDB to debug a 64 bit build of Julia.

Build process is slow/eats memory/hangs my computer

- Disable the Windows Superfetch and Program Compatibility Assistant services, as they are known to have spurious interactions with MinGW/Cygwin.

  As mentioned in the link above: excessive memory use by `svchost` specifically may be investigated in the Task Manager by clicking on the high-memory `svchost.exe` process and selecting `Go to Services`. Disable child services one-by-one until a culprit is found.

- Beware of BLODA. The vmmap tool is indispensable for identifying such software conflicts. Use vmmap to inspect the list of loaded DLLs for bash, mintty, or another persistent process used to drive the build. Essentially *any* DLL outside of the Windows System directory is potential BLODA.

110.5 FreeBSD

Clang is the default compiler on FreeBSD 11.0-RELEASE and above. The remaining build tools are available from the Ports Collection, and can be installed using `pkg install git gcc gmake cmake pkgconf`. To build Julia, simply run `gmake`. (Note that `gmake` must be used rather than `make`, since `make` on FreeBSD corresponds to the incompatible BSD Make rather than GNU Make.)

As mentioned above, it is important to note that the `USE_SYSTEM_*` flags should be used with caution on FreeBSD. This is because many system libraries, and even libraries from the Ports Collection, link to the system's `libgcc_s.so.1`, or to another library which links to the system `libgcc_s`. This library declares its GCC version to be 4.6, which is too old to build Julia, and conflicts with other libraries when linking. Thus it is highly recommended to simply allow Julia to build all of its dependencies. If you do choose to use the `USE_SYSTEM_*` flags, note that `/usr/local` is not on the compiler path by default, so you may need to add `LDFLAGS=-L/usr/local/lib` and `CPPFLAGS=-I/usr/local/include` to your `Make.user`, though doing so may interfere with other dependencies.

Note that the x86 architecture does not support threading due to lack of compiler runtime library support, so you may need to set `JULIA_THREADS=0` in your `Make.user` if you're on a 32-bit system.

110.6 ARM (Linux)

Julia fully supports ARMv8 (AArch64) processors, and supports ARMv7 and ARMv6 (AArch32) with some caveats. This file provides general guidelines for compilation, in addition to instructions for specific devices.

A list of known issues for ARM is available. If you encounter difficulties, please create an issue including the output from `cat /proc/cpuinfo`.

32-bit (ARMv6, ARMv7)

Julia has been successfully compiled on several variants of the following ARMv6 & ARMv7 devices:

- ARMv7 / Cortex A15 Samsung Chromebooks running Ubuntu Linux under Crouton;
- Raspberry Pi.
- Odroid.

Julia requires at least the `armv6` and `vfpv2` instruction sets. It's recommended to use `armv7-a`. `armv5` or soft float are not supported.

Raspberry Pi 1 / Raspberry Pi Zero

If the type of ARM CPU used in the Raspberry Pi is not detected by LLVM, then explicitly set the CPU target by adding the following to `Make.user`:

```
JULIA_CPU_TARGET=arm1176jzf-s
```

To complete the build, you may need to increase the swap file size. To do so, edit `/etc/dphys-swapfile`, changing the line:

```
CONF_SWAPSIZE=100
```

to:

```
CONF_SWAPSIZE=512
```

before restarting the swapfile service:

```
sudo /etc/init.d/dphys-swapfile stop
sudo /etc/init.d/dphys-swapfile start
```

Raspberry Pi 2

The type of ARM CPU used in the Raspberry Pi 2 is not detected by LLVM. Explicitly set the CPU target by adding the following to `Make.user`:

`JULIA_CPU_TARGET=cortex-a7`

Depending on the exact compiler and distribution, there might be a build failure due to unsupported inline assembly. In that case, add `MCPU=armv7-a` to `Make.user`.

AArch64 (ARMv8)

Julia is expected to work and build on ARMv8 cpus. One should follow the general build instructions. Julia expects to have around 8GB of ram or swap enabled to build itself.

Known issues

Starting from Julia v1.10, JITLink is automatically enabled on this architecture for all operating systems when linking to LLVM 15 or later versions. Due to a bug in LLVM memory manager, non-trivial workloads may generate too many memory mappings that on Linux can exceed the limit of memory mappings (`mmap`) set in the file `/proc/sys/vm/max_map_count`, resulting in an error like

```
JIT session error: Cannot allocate memory
```

Should this happen, ask your system administrator to increase the limit of memory mappings for example with the command

```
sysctl -w vm.max_map_count=262144
```

110.7 RISC-V (Linux)

Julia has experimental support for 64-bit RISC-V (RV64) processors running Linux. This file provides general guidelines for compilation, in addition to instructions for specific devices.

A list of known issues for RISC-V is available. If you encounter difficulties, please create an issue including the output from `cat /proc/cpuinfo`.

Compiling Julia

To compilie Julia for RISC-V, you need to manually indicate what architecture, and optionally which CPU to build for. This can be done by setting the `MARCH` and `MCPU` variables in `Make.user`

The `MARCH` variable needs to be set to a RISC-V ISA string, which can be found by looking at the documentation of your device, or by inspecting `/proc/cpuinfo`. Only use flags that your compiler supports, e.g., run `gcc -march=help` to see a list of supported flags. A common value is `rv64gc`, which is a good starting point.

The `MCPU` variable is optional, and can be used to further optimize the generated code for a specific CPU. If you are unsure, it is recommended to leave it unset. You can find a list of supported values by running `gcc --target-help`.

For example, if you are using a StarFive VisionFive2, which contains a JH7110 processor based on the SiFive U74, you can set these flags as follows:

```
MARCH := rv64gc_zba_zbb
MCPU := sifive-u74
```

If you prefer a portable build, you could use:

```
MARCH := rv64gc

# also set JULIA_CPU_TARGET to the expanded form of rv64gc
# (it normally copies the value of MCPU, which we don't set)
JULIA_CPU_TARGET := generic-rv64,i,m,a,f,d,zicsr,zifencei,c
```

Cross-compilation

A native build on a RISC-V device may take a very long time, so it's also possible to cross-compile Julia on a faster machine.

First, get a hold of a RISC-V cross-compilation toolchain that provides support for C, C++ and Fortran. This can be done by checking-out the riscv-gnu-toolchain repository and building it as follows:

```
sudo mkdir /opt/riscv && sudo chown $USER /opt/riscv
./configure --prefix=/opt/riscv --with-languages=c,c++,fortran
make linux -j$(nproc)
```

Then, install the QEMU user-mode emulator for RISC-V, along with `binfmt` support to enable execution of RISC-V binaries on the host machine. The exact steps depend on your distribution, e.g., on Arch Linux it involves installing the `qemu-user-static` and `qemu-user-static-binfmt` packages. Note that to actually

execute RISC-V binaries, QEMU will need to be able to find the RISC-V system root, which can be achieved by setting the `QEMU_LD_PREFIX` environment variable to the path of the root filesystem.

Finally, compile Julia with the following `Make.user` variables (in addition to the ones from the previous section):

```
XC_HOST=riscv64-unknown-linux-gnu
OS=Linux
export QEMU_LD_PREFIX=/opt/riscv/sysroot
```

Note that you will have to execute `make` with `PATH` set to include the cross-compilation toolchain, e.g., by running:

```
PATH=/opt/riscv/bin:$PATH make -j$(nproc)
```

Because of the RISC-V sysroot we use being very barren, you may need to add additional libraries that the Julia build system currently expects to be available system-wide. For example, the build currently relies on a system-provided `libz`, so you may need to copy this library from the Julia build into the system root:

```
make -C deps install-zlib
cp -v usr/lib/libz.*   /opt/riscv/sysroot/usr/lib
cp -v usr/include/z*.h /opt/riscv/sysroot/usr/include
```

110.8 Binary distributions

These notes are for those wishing to compile a binary distribution of Julia for distribution on various platforms. We love users spreading Julia as far and wide as they can, trying it out on as wide an array of operating systems and hardware configurations as possible. As each platform has specific gotchas and processes that must be followed in order to create a portable, working Julia distribution, we have separated most of the notes by OS.

Note that while the code for Julia is MIT-licensed, with a few exceptions, the distribution created by the techniques described herein will be GPL licensed, as various dependent libraries such as `SuiteSparse` are GPL licensed. We do hope to have a non-GPL distribution of Julia in the future.

Versioning and Git

The Makefile uses both the `VERSION` file and commit hashes and tags from the git repository to generate the `base/version_git.jl` with information we use to fill the splash screen and the `versioninfo()` output. If you for some reason don't want to have the git repository available when building you should pregenerate the `base/version_git.jl` file with:

```
make -C base version_git.jl.phony
```

Julia has lots of build dependencies where we use patched versions that has not yet been included by the popular package managers. These dependencies will usually be automatically downloaded when you build, but if you want to be able to build Julia on a computer without internet access you should create a full-source-dist archive with the special make target

```
make full-source-dist
```

that creates a julia-version-commit.tar.gz archive with all required dependencies.

When compiling a tagged release in the git repository, we don't display the branch/commit hash info in the splash screen. You can use this line to show a release description of up to 45 characters. To set this line you have to create a Make.user file containing:

```
override TAGGED_RELEASE_BANNER = "my-package-repository build"
```

Target Architectures

By default, Julia optimizes its system image to the native architecture of the build machine. This is usually not what you want when building packages, as it will make Julia fail at startup on any machine with incompatible CPUs (in particular older ones with more restricted instruction sets).

We therefore recommend that you pass the `MARCH` variable when calling `make`, setting it to the baseline target you intend to support. This will determine the target CPU for both the Julia executable and libraries, and the system image (the latter can also be set using `JULIA_CPU_TARGET`). Typically useful values for x86 CPUs are `x86-64` and `core2` (for 64-bit builds) and `pentium4` (for 32-bit builds). Unfortunately, CPUs older than Pentium 4 are currently not supported (see this issue).

The full list of CPU targets supported by LLVM can be obtained by running `llc -mattr=help`.

Linux

On Linux, `make binary-dist` creates a tarball that contains a fully functional Julia installation. If you wish to create a distribution package such as a `.deb`, or `.rpm`, some extra effort is needed. See the julia-debian repository for an example of what metadata is needed for creating `.deb` packages for Debian and Ubuntu-based systems. See the Fedora package for RPM-based distributions. Although we have not yet experimented with it, Alien could be used to generate Julia packages for various Linux distributions.

Julia supports overriding standard installation directories via `prefix` and other environment variables you can pass when calling `make` and `make install`. See Make.inc for their list. `DESTDIR` can also be used to force the installation into a temporary directory.

By default, Julia loads `$prefix/etc/julia/startup.jl` as an installation-wide initialization file. This file can be used by distribution managers to set up custom paths or initialization code. For Linux distribution packages, if `$prefix` is set to `/usr`, there is no `/usr/etc` to look into. This requires the path to Julia's private `etc` directory to be changed. This can be done via the `sysconfdir` make variable when building. Simply pass `sysconfdir=/etc` to `make` when building and Julia will first check `/etc/julia/startup.jl` before trying `$prefix/etc/julia/startup.jl`.

OS X

To create a binary distribution on OSX, build Julia first, then cd to `contrib/mac/app`, and run `make` with the same makevars that were used with `make` when building Julia proper. This will then create a `.dmg` file in the `contrib/mac/app` directory holding a completely self-contained Julia.app.

Alternatively, Julia may be built as a framework by invoking `make` with the `darwinframework` target and `DARWIN_FRAMEWORK=1` set. For example, `make DARWIN_FRAMEWORK=1 darwinframework`.

Windows

Instructions for creating a Julia distribution on Windows are described in the build devdocs for Windows.

Notes on BLAS and LAPACK

Julia builds OpenBLAS by default, which includes the BLAS and LAPACK libraries. On 32-bit architectures, Julia builds OpenBLAS to use 32-bit integers, while on 64-bit architectures, Julia builds OpenBLAS to use 64-bit integers (ILP64). It is essential that all Julia functions that call BLAS and LAPACK API routines use integers of the correct width.

Most BLAS and LAPACK distributions provided on linux distributions, and even commercial implementations ship libraries that use 32-bit APIs. In many cases, a 64-bit API is provided as a separate library.

When using vendor provided or OS provided libraries, a `make` option called `USE_BLAS64` is available as part of the Julia build. When doing `make USE_BLAS64=0`, Julia will call BLAS and LAPACK assuming a 32-bit API, where all integers are 32-bit wide, even on a 64-bit architecture.

Other libraries that Julia uses, such as SuiteSparse also use BLAS and LAPACK internally. The APIs need to be consistent across all libraries that depend on BLAS and LAPACK. The Julia build process will build all these libraries correctly, but when overriding defaults and using system provided libraries, this consistency must be ensured.

Also note that Linux distributions sometimes ship several versions of OpenBLAS, some of which enable multithreading, and others only working in a serial fashion. For example, in Fedora, `libopenblasp.so` is threaded, but `libopenblas.so` is not. We recommend using the former for optimal performance. To choose an OpenBLAS library whose name is different from the default `libopenblas.so`, pass `LIBBLAS=-l$(YOURBLAS)` and `LIBBLASNAME=lib$(YOURBLAS)` to `make`, replacing `$(YOURBLAS)` with the name of your library. You can also add `.so.0` to the name of the library if you want your package to work without requiring the unversioned `.so` symlink.

Finally, OpenBLAS includes its own optimized version of LAPACK. If you set `USE_SYSTEM_BLAS=1` and `USE_SYSTEM_LAPACK=1`, you should also set `LIBLAPACK=-l$(YOURBLAS)` and `LIBLAPACKNAME=lib$(YOURBLAS)`. Else, the reference LAPACK will be used and performance will typically be much lower.

Starting with Julia 1.7, Julia uses libblastrampoline to pick a different BLAS at runtime.

110.9 Point releasing 101

Creating a point/patch release consists of several distinct steps.

Backporting commits

Some pull requests are labeled "backport pending x.y", e.g. "backport pending 0.6". This designates that the next subsequent release tagged from the release-x.y branch should include the commit(s) in that pull request. Once the pull request is merged into master, each of the commits should be cherry picked to a dedicated branch that will ultimately be merged into release-x.y.

Creating a backports branch

First, create a new branch based on release-x.y. The typical convention for Julia branches is to prefix the branch name with your initials if it's intended to be a personal branch. For the sake of example, we'll say that the author of the branch is Jane Smith.

```
git fetch origin
git checkout release-x.y
git rebase origin/release-x.y
git checkout -b js/backport-x.y
```

This ensures that your local copy of release-x.y is up to date with origin before you create a new branch from it.

Cherry picking commits

Now we do the actual backporting. Find all merged pull requests labeled "backport pending x.y" in the GitHub web UI. For each of these, scroll to the bottom where it says "someperson merged commit 123abc into `master` XX minutes ago". Note that the commit name is a link; if you click it, you'll be shown the contents of the commit. If this page shows that 123abc is a merge commit, go back to the PR page—we don't want merge commits, we want the actual commits. However, if this does not show a merge commit, it means that the PR was squash-merged. In that case, use the git SHA of the commit, listed next to commit on this page.

Once you have the SHA of the commit, cherry-pick it onto the backporting branch:

```
git cherry-pick -x -e <sha>
```

There may be conflicts which need to be resolved manually. Once conflicts are resolved (if applicable), add a reference to the GitHub pull request that introduced the commit in the body of the commit message.

After all of the relevant commits are on the backports branch, push the branch to GitHub.

Checking for performance regressions

Point releases should never introduce performance regressions. Luckily the Julia benchmarking bot, Nanosoldier, can run benchmarks against any branch, not just master. In this case we want to check the benchmark results of js/backport-x.y against release-x.y. To do this, awaken the Nanosoldier from his robotic slumber using a comment on your backporting pull request:

```
@nanosoldier `runbenchmarks(ALL, vs=":release-x.y")`
```

This will run all registered benchmarks on release-x.y and js/backport-x.y and produce a summary of results, marking all improvements and regressions.

If Nanosoldier finds any regressions, try verifying locally and rerun Nanosoldier if necessary. If the regressions are deemed to be real rather than just noise, you'll have to find a commit on master to backport that fixes it if one exists, otherwise you should determine what caused the regression and submit a patch (or get someone who knows the code to submit a patch) to master, then backport the commit once that's merged. (Or submit a patch directly to the backport branch if appropriate.)

Building test binaries

After the backport PR has been merged into the `release-x.y` branch, update your local clone of Julia, then get the SHA of the branch using

```
git rev-parse origin/release-x.y
```

Keep that handy, as it's what you'll enter in the "Revision" field in the buildbot UI.

For now, all you need are binaries for Linux x86-64, since this is what's used for running PackageEvaluator. Go to https://buildog.julialang.org, submit a job for `nuke_linux64`, then queue up a job for `package_linux64`, providing the SHA as the revision. When the packaging job completes, it will upload the binary to the `julialang2` bucket on AWS. Retrieve the URL, as it will be used for PackageEvaluator.

Checking for package breakages

Point releases should never break packages, with the possible exception of packages that are doing some seriously questionable hacks using Base internals that are not intended to be user-facing. (In those cases, maybe have a word with the package author.)

Checking whether changes made in the forthcoming new version will break packages can be accomplished using PackageEvaluator, often called "PkgEval" for short. PkgEval is what populates the status badges on GitHub repos and on pkg.julialang.org. It typically runs on one of the non-benchmarking nodes of Nanosoldier and uses Vagrant to perform its duties in separate, parallel VirtualBox virtual machines.

Setting up PackageEvaluator

Clone PackageEvaluator and create a branch called `backport-x.y.z`, and check it out. Note that the required changes are a little hacky and confusing, and hopefully that will be addressed in a future version of PackageEvaluator. The changes to make will be modeled off of this commit.

The setup script takes its first argument as the version of Julia to run and the second as the range of package names (AK for packages named A-K, LZ for L-Z). The basic idea is that we're going to tweak that a bit to run only two versions of Julia, the current x.y release and our backport version, each with three ranges of packages.

In the linked diff, we're saying that if the second argument is LZ, use the binaries built from our backport branch, otherwise (AK) use the release binaries. Then we're using the first argument to run a section of the package list: A-F for input 0.4, G-N for 0.5, and O-Z for 0.6.

Running PackageEvaluator

To run PkgEval, find a hefty enough machine (such as Nanosoldier node 1), then run

```
git clone https://github.com/JuliaCI/PackageEvaluator.jl.git
cd PackageEvaluator.jl/scripts
git checkout backport-x.y.z
./runvagrant.sh
```

This produces some folders in the scripts/ directory. The folder names and their contents are decoded below:

Investigating results

Once that's done, you can use `./summary.sh` from that same directory to produce a summary report of the findings. We'll do so for each of the folders to aggregate overall results by version.

Folder name	Julia version	Package range
0.4AK	Release	A-F
0.4LZ	Backport	A-F
0.5AK	Release	G-N
0.5LZ	Backport	G-N
0.6AK	Release	O-Z
0.6LZ	Backport	O-Z

```
./summary.sh 0.4AK/*.json > summary_release.txt
./summary.sh 0.5AK/*.json >> summary_release.txt
./summary.sh 0.6AK/*.json >> summary_release.txt
./summary.sh 0.4LZ/*.json > summary_backport.txt
./summary.sh 0.5LZ/*.json >> summary_backport.txt
./summary.sh 0.6LZ/*.json >> summary_backport.txt
```

Now we have two files, `summary_release.txt` and `summary_backport.txt`, containing the PackageEvaluator test results (pass/fail) for each package for the two versions.

To make these easier to ingest into a Julia, we'll convert them into CSV files then use the DataFrames package to process the results. To convert to CSV, copy each .txt file to a corresponding .csv file, then enter Vim and execute `ggVGI"<esc>` then `:%s/\.json /",/g`. (You don't have to use Vim; this just is one way to do it.) Now process the results with Julia code similar to the following.

```
using DataFrames

release = readtable("summary_release.csv", header=false, names=[:package, :release])
backport = readtable("summary_backport.csv", header=false, names=[:package, :backport])

results = join(release, backport, on=:package, kind=:outer)

for result in eachrow(results)
    a = result[:release]
    b = result[:backport]
    if (isna(a) && !isna(b)) || (isna(b) && !isna(a))
        color = :yellow
    elseif a != b && occursin("pass", b)
        color = :green
    elseif a != b
        color = :red
    else
        continue
    end
    printstyled(result[:package], ": Release ", a, " -> Backport ", b, "\n", color=color)
end
```

This will write color-coded lines to `stdout`. All lines in red must be investigated as they signify potential breakages caused by the backport version. Lines in yellow should be looked into since it means a package ran on one version but not on the other for some reason. If you find that your backported branch is causing breakages, use `git bisect` to identify the problematic commits, `git revert` those commits, and repeat the process.

Merging backports into the release branch

After you have ensured that

- the backported commits pass all of Julia's unit tests,
- there are no performance regressions introduced by the backported commits as compared to the release branch, and
- the backported commits do not break any registered packages,

then the backport branch is ready to be merged into release-x.y. Once it's merged, go through and remove the "backport pending x.y" label from all pull requests containing the commits that have been backported. Do not remove the label from PRs that have not been backported.

The release-x.y branch should now contain all of the new commits. The last thing we want to do to the branch is to adjust the version number. To do this, submit a PR against release-x.y that edits the VERSION file to remove `-pre` from the version number. Once that's merged, we're ready to tag.

Tagging the release

It's time! Check out the release-x.y branch and make sure that your local copy of the branch is up to date with the remote branch. At the command line, run

```
git tag v$(cat VERSION)
git push --tags
```

This creates the tag locally and pushes it to GitHub.

After tagging the release, submit another PR to release-x.y to bump the patch number and add `-pre` back to the end. This denotes that the branch state reflects a prerelease version of the next point release in the x.y series.

Follow the remaining directions in the Makefile.

Signing binaries

Some of these steps will require secure passwords. To obtain the appropriate passwords, contact Elliot Saba (staticfloat) or Alex Arslan (ararslan). Note that code signing for each platform must be performed on that platform (e.g. Windows signing must be done on Windows, etc.).

Linux

Code signing must be done manually on Linux, but it's quite simple. First obtain the file `julia.key` from the CodeSigning folder in the `juliasecure` AWS bucket. Add this to your GnuPG keyring using

```
gpg --import julia.key
```

This will require entering a password that you must obtain from Elliot or Alex. Next, set the trust level for the key to maximum. Start by entering a gpg session:

```
gpg --edit-key julia
```

At the prompt, type `trust`, then when asked for a trust level, provide the maximum available (likely 5). Exit GnuPG.

Now, for each of the Linux tarballs that were built on the buildbots, enter

```
gpg -u julia --armor --detach-sig julia-x.y.z-linux-<arch>.tar.gz
```

This will produce a corresponding .asc file for each tarball. And that's it!

macOS

Code signing should happen automatically on the macOS buildbots. However, it's important to verify that it was successful. On a system or virtual machine running macOS, download the .dmg file that was built on the buildbots. For the sake of example, say that the .dmg file is called `julia-x.y.z-osx.dmg`. Run

```
mkdir ./jlmnt
hdiutil mount -readonly -mountpoint ./jlmnt julia-x.y.z-osx.dmg
codesign -v jlmnt/Julia-x.y.app
```

Be sure to note the name of the mounted disk listed when mounting! For the sake of example, we'll assume this is `disk3`. If the code signing verification exited successfully, there will be no output from the `codesign` step. If it was indeed successful, you can detach the .dmg now:

```
hdiutil eject /dev/disk3
rm -rf ./jlmnt
```

If you get a message like

> Julia-x.y.app: code object is not signed at all

then you'll need to sign manually.

To sign manually, first retrieve the OS X certificates from the CodeSigning folder in the `juliasecure` bucket on AWS. Add the .p12 file to your keychain using Keychain.app. Ask Elliot Saba (staticfloat) or Alex Arslan (ararslan) for the password for the key. Now run

```
hdiutil convert julia-x.y.z-osx.dmg -format UDRW -o julia-x.y.z-osx_writable.dmg
mkdir ./jlmnt
hdiutil mount -mountpoint julia-x.y.z-osx_writable.dmg
codesign -s "AFB379C0B4CBD9DB9A762797FC2AB5460A2B0DBE" --deep jlmnt/Julia-x.y.app
```

This may fail with a message like

> Julia-x.y.app: resource fork, Finder information, or similar detritus not allowed

If that's the case, you'll need to remove extraneous attributes:

```
xattr -cr jlmnt/Julia-x.y.app
```

Then retry code signing. If that produces no errors, retry verification. If all is now well, unmount the writable .dmg and convert it back to read-only:

```
hdiutil eject /dev/disk3
rm -rf ./jlmnt
hdiutil convert julia-x.y.z-osx_writable.dmg -format UDZO -o julia-x.y.z-osx_fixed.dmg
```

Verify that the resulting .dmg is in fact fixed by double clicking it. If everything looks good, eject it then drop the `_fixed` suffix from the name. And that's it!

Windows

Signing must be performed manually on Windows. First obtain the Windows 10 SDK, which contains the necessary signing utilities, from the Microsoft website. We need the `SignTool` utility which should have been installed somewhere like `C:\Program Files (x86)\Windows Kits\10\App Certification Kit`. Grab the Windows certificate files from CodeSigning on `juliasecure` and put them in the same directory as the executables. Open a Windows CMD window, `cd` to where all the files are, and run

```
set PATH=%PATH%;C:\Program Files (x86)\Windows Kits\10\App Certification Kit;
signtool sign /f julia-windows-code-sign_2017.p12 /p "PASSWORD" ^
   /t http://timestamp.verisign.com/scripts/timstamp.dll ^
   /v julia-x.y.z-win32.exe
```

Note that `^` is a line continuation character in Windows CMD and `PASSWORD` is a placeholder for the password for this certificate. As usual, contact Elliot or Alex for passwords. If there are no errors, we're all good!

Uploading binaries

Now that everything is signed, we need to upload the binaries to AWS. You can use a program like Cyberduck or the `aws` command line utility. The binaries should go in the `julialang2` bucket in the appropriate folders. For example, Linux x86-64 goes in `julialang2/bin/linux/x.y`. Be sure to delete the current `julia-x.y-latest-linux-<arch>.tar.gz` file and replace it with a duplicate of `julia-x.y.z-linux-<arch>.tar.gz`.

We also need to upload the checksums for everything we've built, including the source tarballs and all release binaries. This is simple:

```
shasum -a 256 julia-x.y.z* | grep -v -e sha256 -e md5 -e asc > julia-x.y.z.sha256
md5sum julia-x.y.z* | grep -v -e sha256 -e md5 -e asc > julia-x.y.z.md5
```

Note that if you're running those commands on macOS, you'll get very slightly different output, which can be reformatted by looking at an existing file. Mac users will also need to use `md5 -r` instead of `md5sum`. Upload the .md5 and .sha256 files to `julialang2/bin/checksums` on AWS.

Ensure that the permissions on AWS for all uploaded files are set to "Everyone: READ."

For each file we've uploaded, we need to purge the Fastly cache so that the links on the website point to the updated files. As an example:

```
curl -X PURGE https://julialang-s3.julialang.org/bin/checksums/julia-x.y.z.sha256
```

Sometimes this isn't necessary but it's good to do anyway.

Chapter 111

Contributor's Guide

111.1 Code changes

Contributing to core functionality or base libraries

By contributing code to Julia, you are agreeing to release it under the MIT License.

The Julia community uses GitHub issues to track and discuss problems, feature requests, and pull requests (PR).

Issues and pull requests should have self explanatory titles such that they can be understood from the list of PRs and Issues. i.e. `Add {feature}` and `Fix {bug}` are good, `Fix #12345. Corrects the bug.` is bad.

You can make pull requests for incomplete features to get code review. The convention is to open these as draft PRs and prefix the pull request title with "WIP:" for Work In Progress, or "RFC:" for Request for Comments when work is completed and ready for merging. This will prevent accidental merging of work that is in progress.

Note: These instructions are for adding to or improving functionality in the base library. Before getting started, it can be helpful to discuss the proposed changes or additions on the Julia Discourse forum or in a GitHub issue—it's possible your proposed change belongs in a package rather than the core language. Also, keep in mind that changing stuff in the base can potentially break a lot of things. Finally, because of the time required to build Julia, note that it's usually faster to develop your code in stand-alone files, get it working, and then migrate it into the base libraries.

Add new code to Julia's base libraries as follows (this is the "basic" approach; see a more efficient approach in the next section):

1. Edit the appropriate file in the `base/` directory, or add new files if necessary. Create tests for your functionality and add them to files in the `test/` directory. If you're editing C or Scheme code, most likely it lives in `src/` or one of its subdirectories, although some aspects of Julia's REPL initialization live in `cli/`.
2. Add any new files to `sysimg.jl` in order to build them into the Julia system image.
3. Add any necessary export symbols in `exports.jl`.
4. Include your tests in `test/Makefile` and `test/choosetests.jl`.

Build as usual, and do `make clean testall` to test your contribution. If your contribution includes changes to Makefiles or external dependencies, make sure you can build Julia from a clean tree using `git clean -fdx` or equivalent (be careful – this command will delete any files lying around that aren't checked into git).

Running specific tests

There are `make` targets for running specific tests:

```
make test-bitarray
```

You can also use the `runtests.jl` script, e.g. to run `test/bitarray.jl` and `test/math.jl`:

```
./usr/bin/julia test/runtests.jl bitarray math
```

Modifying base more efficiently with Revise.jl

Revise is a package that tracks changes in source files and automatically updates function definitions in your running Julia session. Using it, you can make extensive changes to Base without needing to rebuild in order to test your changes.

Here is the standard procedure:

1. If you are planning changes to any types or macros, make those changes and build julia using `make`. (This is necessary because `Revise` cannot handle changes to type definitions or macros.) Unless it's required to get Julia to build, you do not have to add any functionality based on the new types, just the type definitions themselves.
2. Start a Julia REPL session. Then issue the following commands:

```
using Revise    # if you aren't launching it in your `.julia/config/startup.jl`
Revise.track(Base)
```

1. Edit files in `base/`, save your edits, and test the functionality.

If you need to restart your Julia session, just start at step 2 above. `Revise.track(Base)` will note any changes from when Julia was last built and incorporate them automatically. You only need to rebuild Julia if you made code-changes that Revise cannot handle.

For convenience, there are also `test-revise-*` targets for every `test-*` target that use Revise to load any modifications to Base into the current system image before running the corresponding test. This can be useful as a shortcut on the command line (since tests aren't always designed to be run outside the runtest harness).

Contributing to the standard library

The standard library (stdlib) packages are baked into the Julia system image. When running the ordinary test workflow on the stdlib packages, the system image version overrides the version you are developing. To test stdlib packages, you can do the following steps:

1. Edit the UUID field of the `Project.toml` in the stdlib package
2. Change the current directory to the directory of the stdlib you are developing
3. Start julia with `julia --project=.`
4. You can now test the package by running `pkg> test` in Pkg mode.

Because you changed the UUID, the package manager treats the stdlib package as different from the one in the system image, and the system image version will not override the package.

Be sure to change the UUID value back before making the pull request.

News-worthy changes

For new functionality and other substantial changes, add a brief summary to `NEWS.md`. The news item should cross reference the pull request (PR) parenthetically, in the form `([#pr])`. To add the PR reference number, first create the PR, then push an additional commit updating `NEWS.md` with the PR reference number. We periodically run `./julia doc/NEWS-update.jl` from the julia directory to update the cross-reference links, but this should not be done in a typical PR in order to avoid conflicting commits.

Annotations for new features, deprecations and behavior changes

API additions and deprecations, and minor behavior changes are allowed in minor version releases. For documented features that are part of the public API, a compatibility note should be added into the manual or the docstring. It should state the Julia minor version that changed the behavior and have a brief message describing the change.

At the moment, this should always be done with the following `compat` admonition (so that it would be possible to programmatically find the annotations in the future):

```
!!! compat "Julia 1.X" This method was added in Julia 1.X.
```

111.2 Writing tests

There are never enough tests. Track code coverage at Codecov, and help improve it.

1. Go visit https://codecov.io/github/JuliaLang/julia.
2. Browse through the source files and find some untested functionality (highlighted in red) that you think you might be able to write a test for.
3. Write a test that exercises this functionality—you can add your test to one of the existing files, or start a new one, whichever seems most appropriate to you. If you're adding a new test file, make sure you include it in the list of tests in `test/choosetests.jl`. https://docs.julialang.org/en/v1/stdlib/Test/ may be helpful in explaining how the testing infrastructure works.

4. Run `make test-all` to rebuild Julia and run your new test(s). If you had to fix a bug or add functionality in `base`, this will ensure that your test passes and that you have not introduced extraneous whitespace.
5. Submit the test as a pull request (PR).

- Code for the buildbot configuration is maintained at: https://github.com/staticfloat/julia-buildbot
- You can see the current buildbot setup at: https://build.julialang.org/builders
- Issue 9493 and issue 11885 have more detailed discussion on code coverage.

Code coverage shows functionality that still needs "proof of concept" tests. These are important, as are tests for tricky edge cases, such as converting between integer types when the number to convert is near the maximum of the range of one of the integer types. Even if a function already has some coverage on Codecov, it may still benefit from tests for edge cases.

111.3 Improving documentation

By contributing documentation to Julia, you are agreeing to release it under the MIT License.

Julia's documentation source files are stored in the `doc/` directory and all docstrings are found in `base/`. Like everything else these can be modified using `git`. Documentation is built with Documenter.jl, which uses Markdown syntax. The HTML documentation can be built locally by running

```
make docs
```

from Julia's root directory. This will rebuild the Julia system image, then install or update the package dependencies required to build the documentation, and finally build the HTML documentation and place the resulting files in `doc/_build/html/`.

> **Note**
>
> When making changes to any of Julia's documentation it is recommended that you run `make docs` to check that your changes are valid and do not produce any errors before opening a pull request.

Below are outlined the three most common types of documentation changes and the steps required to perform them. Please note that the following instructions do not cover the full range of features provided by Documenter.jl. Refer to Documenter's documentation if you encounter anything that is not covered by the sections below.

Modifying files in `doc/src/`

Most of the source text for the Julia Manual is located in `doc/src/`. To update or add new text to any one of the existing files the following steps should be followed:

1. update the text in whichever `.md` files are applicable;
2. run `make docs` from the root directory;

3. check the output in `doc/_build/html/` to make sure the changes are correct;
4. commit your changes and open a pull request.

   Note

   The contents of `doc/_build/` does **not** need to be committed when you make changes.

To add a **new file** to `doc/src/` rather than updating a file replace step 1 above with

1. add the file to the appropriate subdirectory in `doc/src/` and also add the file path to the PAGES vector in `doc/make.jl`.

Modifying an existing docstring in `base/`

All docstrings are written inline above the methods or types they are associated with and can be found by clicking on the `source` link that appears below each docstring in the HTML file. The steps needed to make a change to an existing docstring are listed below:

1. find the docstring in `base/`;
2. update the text in the docstring;
3. run `make docs` from the root directory;
4. check the output in `doc/_build/html/` to make sure the changes are correct;
5. commit your changes and open a pull request.

Adding a new docstring to `base/`

The steps required to add a new docstring are listed below:

1. find a suitable definition in `base/` that the docstring will be most applicable to;
2. add a docstring above the definition;
3. find a suitable `@docs` code block in one of the `doc/src/stdlib/` files where you would like the docstring to appear;
4. add the name of the definition to the `@docs` code block. For example, with a docstring added to a function `bar`

   ```
   julia "..." function bar(args...) # ... end
   ```

   you would add the name `bar` to a `@docs` block in `doc/src/stdlib/`

   ````
   ```@docs
   foo
   bar # <-- Added this one.
   baz
   ```
   ````

5. run `make docs` from the root directory;
6. check the output in `doc/_build/html` to make sure the changes are correct;
7. commit your changes and open a pull request.

Doctests

Examples written within docstrings can be used as testcases known as "doctests" by annotating code blocks with `jldoctest`.

````
```jldoctest
julia> uppercase("Docstring test")
"DOCSTRING TEST"
```
````

A doctest needs to match an interactive REPL including the `julia>` prompt. It is recommended to add the header `# Examples` above the doctests.

See the documentation of writing jldoctests for best practices on how to write doctests for common scenarios and the `doc/README.md` file for how to run the doctests.

111.4 Writing jldoctests

This page describes how to write and maintain `jldoctest` blocks in the documentation. Following these guidelines helps keep doctests reliable and easy to read.

Filters

Use `filter =` whenever output contains text that might vary across runs. The following are common situations where this may happen:

- The output contains arrays with undefined memory (e.g. from `undef` or `similar`)
- The output contains random numbers
- The output contains timing information
- The output contains file system paths

Common filter sequences

The documentation relies on several recurring patterns:

- `r"int.jl:\\d+"` — remove line numbers from introspection macros.
- `r"Stacktrace:(\\n \\[0-9]+\\].*)*"` — hide stack traces when illustrating errors.
- `r"Closest candidates.*\\n .*"` — skip the method suggestions printed by `MethodError`.
- `r"@ .*"` — strip file locations from the output of `methods` or `@which`.
- `r"\\@world\\(MyStruct, \\d+:\\d+\\)"` — filter world age numbers.
- `r"with \\d+ methods"` — ignore method counts when redefining functions.
- `r"[0-9\\.]+ seconds \\(.*?\\)"` — remove timing output with memory information.
- `r"[0-9\\.]+ seconds"` — remove simple timing results.

- `r"[0-9\\.]+"` — filter digits from names such as anonymous functions.
- `r"([A-B] [0-5])"` and `r"[A-B] [X-Z] [0-5]"` — account for non-deterministic process output.
- `r"(world\\nhello|hello\\nworld)"` — allow either ordering of interleaved output.

If none of these match your situation, craft a regular expression that removes the varying text. Using filters keeps doctests stable across platforms and Julia versions.

Double escaping in docstrings

When writing regex filters inside docstrings, remember to double escape backslashes. For example, use `r"[\\d\\.]+"` instead of `r"[\d\.]+"`. This is necessary because the docstring itself processes escape sequences before the regex is created.

Setup code

Small setup expressions may be placed inline using the `setup =` option:

````
```jldoctest; setup = :(using InteractiveUtils)
...
```
````

For longer setup code or if multiple blocks require the same environment, use the `DocTestSetup` meta block:

````
```@meta
DocTestSetup = :(import Random; Random.seed!(1234))
```
````

and disable it afterwards with

````
```@meta
DocTestSetup = nothing
```
````

Teardown code

If you need teardown code (e.g. to delete created temporary files or to reset the current directory), you can use the `teardown =` option:

````
```jldoctest; setup = :(oldpath = pwd(); cd(mktempdir())), teardown = :(cd(oldpath))
...
```
````

Maintaining state between snippets

Related doctest blocks can share state by giving them the same label after the `jldoctest` keyword. The manual uses this pattern to demonstrate mutation:

````
```jldoctest mutation_vs_rebind
julia> a = [1,2,3]
...
```
````

and later

````
```jldoctest mutation_vs_rebind
julia> a[1] = 42
...
```
````

Blocks with the same name execute sequentially during doctesting, so variables created in the first block remain available in the following ones.

When a snippet needs to preserve its result for later examples, give it a label and reuse that label. This avoids repeating setup code and mirrors a REPL session more closely.

Further reading

For a complete reference of doctest syntax, see the corresponding Documenter.jl docs.

111.5 Contributing to patch releases

The process of creating a patch release is roughly as follows:

1. Create a new branch (e.g. `backports-release-1.10`) against the relevant minor release branch (e.g. `release-1.10`). Usually a corresponding pull request is created as well.
2. Add commits, nominally from `master` (hence "backports"), to that branch. See below for more information on this process.
3. Run the BaseBenchmarks.jl benchmark suite and PkgEval.jl package ecosystem exerciser against that branch. Nominally BaseBenchmarks.jl and PkgEval.jl are invoked via Nanosoldier.jl from the pull request associated with the backports branch. Fix any issues.
4. Once all test and benchmark reports look good, merge the backports branch into the corresponding release branch (e.g. merge `backports-release-1.10` into `release-1.10`).
5. Open a pull request that bumps the version of the relevant minor release to the next patch version, e.g. as in this pull request.
6. Ping `@JuliaLang/releases` to tag the patch release and update the website.
7. Open a pull request that bumps the version of the relevant minor release to the next prerelease patch version, e.g. as in this pull request.

Step 2 above, i.e. backporting commits to the `backports-release-X.Y` branch, has largely been automated via `Backporter`: Backporter searches for merged pull requests with the relevant `backport-X.Y` tag, and attempts to cherry-pick the commits from those pull requests onto the `backports-release-X.Y` branch. Some commits apply successfully without intervention, others not so much. The latter commits require "manual" backporting, with which help is generally much appreciated. Backporter generates a report identifying those commits it managed to backport automatically and those that require manual backporting; this report is usually copied into the first post of the pull request associated with `backports-release-X.Y` and maintained as additional commits are automatically and/or manually backported.

When contributing a manual backport, if you have the necessary permissions, please push the backport directly to the `backports-release-X.Y` branch. If you lack the relevant permissions, please open a pull request against the `backports-release-X.Y` branch with the manual backport. Once the manual backport is live on the `backports-release-X.Y` branch, please remove the `backport-X.Y` tag from the originating pull request for the commits.

111.6 Code Formatting Guidelines

General Formatting Guidelines for Julia code contributions

- Follow the latest dev version of Julia Style Guide.
- Use whitespace to make the code more readable
- No whitespace at the end of a line (trailing whitespace)
- Comments are good, especially when they explain the algorithm
- Try to adhere to a 92 character line length limit
- It is generally preferred to use ASCII operators and identifiers over Unicode equivalents whenever possible
- In docstrings refer to the language as "Julia" and the executable as "`julia`"

General Formatting Guidelines For C code contributions

- 4 spaces per indentation level, no tabs
- Space between `if` and `(` (`if (x) ...`)
- Newline before opening `{` in function definitions
- `f(void)` for 0-argument function declarations
- Newline between `}` and `else` instead of `} else {`
- If one part of an `if..else` chain uses `{ }` then all should
- No whitespace at the end of a line

111.7 Git workflow recommendations

Git Recommendations For Pull Requests

- Avoid working from the `master` branch of your fork. Create a new branch as it will make it easier to update your pull request if Julia's `master` changes.
- Try to squash together small commits that make repeated changes to the same section of code, so your pull request is easier to review. A reasonable number of separate well-factored commits is fine, especially for larger changes.
- If any conflicts arise due to changes in Julia's `master`, prefer updating your pull request branch with `git rebase` versus `git merge` or `git pull`, since the latter will introduce merge commits that clutter the git history with noise that makes your changes more difficult to review.
- Descriptive commit messages are good.
- Using `git add -p` or `git add -i` can be useful to avoid accidentally committing unrelated changes.
- When linking to specific lines of code in discussion of an issue or pull request, hit the `y` key while viewing code on GitHub to reload the page with a URL that includes the specific version that you're viewing. That way any lines of code that you refer to will still make sense in the future, even if the content of the file changes.
- Whitespace can be automatically removed from existing commits with `git rebase`.
  - To remove whitespace for the previous commit, run `git rebase --whitespace=fix HEAD~1`.
  - To remove whitespace relative to the `master` branch, run `git rebase --whitespace=fix master`.

Git Recommendations For Pull Request Reviewers

- When merging, we generally like `squash+merge`. Unless it is the rare case of a PR with carefully staged individual commits that you want in the history separately, in which case `merge` is acceptable, but usually prefer `squash+merge`.

111.8 Using AI agents to work on Julia

> ![WARNING] You are responsible for the code you submit in PRs. Do not submit PRs containing AI-generated code that you do not understand or that does not meet the ordinary quality bar for PRs to julia.

This page documents best practices for setting up AI agents to work with Julia. If you find additional prompt instructions that work well for common tasks, consider submitting a PR to add these to AGENTS.md.

Google Jules

Use the following for your `Initial Setup` configuration.

```
curl -fsSL https://install.julialang.org | sh -s -- -y --default-channel nightly
. /home/swebot/.profile
```

Jules has access to the internet, so you can give it links to issues or additional documentation in your prompting.

OpenAI Codex

Configure the following:

Setup Script

```
apt update
apt install less
curl -fsSL https://install.julialang.org | sh -s -- -y --default-channel nightly
source /root/.bashrc
make -C /workspace/julia/doc alldeps JULIA_EXECUTABLE="/root/.juliaup/bin/julia"
make -C /workspace/julia/test install-revise-deps JULIA_EXECUTABLE="/root/.juliaup/bin/julia"
```

Environment Variables

```
JULIA_PKG_OFFLINE=true
```

Codex does not have internet access after initial setup, so you cannot give it additional information as links - you will need to copy any relevant text into the prompt.

Note that Codex rebuilds the environment after every invocation. This can add significant latency. Codex work best for well-defined tasks that can be solved in a single shot.

Chapter 112

Julia v1.13 Release Notes

112.1 New language features

- New `@__FUNCTION__` macro to refer to the innermost enclosing function (#58940).
- The character U+1F8B2 🢲 (RIGHTWARDS ARROW WITH LOWER HOOK), newly added by Unicode 16, is now a valid operator with arrow precedence, accessible as `\hookunderrightarrow` at the REPL ([JuliaLang/JuliaSyntax.jl#525], #57143).
- Support for Unicode 17 (#59534).

112.2 Language changes

- The return value of `@doc` has changed: it now returns the documented expression (e.g. a function or type) instead of a `Docs.Binding` object as in previous versions. Code that depended on `@doc` returning a `Docs.Binding` will need to be updated (#59882, #60681).
- The `hash` algorithm and its values have changed for certain types, most notably `AbstractString`. Any `hash` specializations for equal types to those that changed, such as some third-party string packages, may need to be deleted (#57509, #59691).
- The `hash(::AbstractString)` function is now a zero-copy / zero-cost function, based upon providing a correct implementation of the `codeunit` and `iterate` functions. Third-party string packages should migrate to the new algorithm by deleting their existing overrides of the `hash` function (#59691).

112.3 Command-line option changes

- The option `--sysimage-native-code=no` has been deprecated.
- The `JULIA_CPU_TARGET` environment variable now supports a `sysimage` keyword to match (or extend) the CPU target used to build the current system image (#58970).
- The `--code-coverage=all` option now automatically throws away sysimage caches so that code coverage can be accurately measured on methods within the sysimage. It is thrown away after startup (and after startup.jl), before any user code is executed (#59234).

- New `--trace-eval` command-line option to show expressions being evaluated during top-level evaluation. Supports `--trace-eval=loc` or just `--trace-eval` (show location only), `--trace-eval=full` (show full expressions), and `--trace-eval=no` (disable tracing). Also adds `Base.TRACE_EVAL` global control that takes priority over the command-line option and can be set to `:no`, `:loc`, `:full`, or `nothing` (to use command-line setting) (#57137).
- Julia now automatically enables verbose debugging options (`--trace-eval` and `JULIA_TEST_VERBOSE`) when CI debugging has been triggered. i.e. via the "debug logging" UI toggle is enabled on github actions re-runs. Other platforms are supported too (#59551).

112.4 Multi-threading changes

- A new `AbstractSpinLock` is defined with `SpinLock <: AbstractSpinLock` (#55944).
- A new `PaddedSpinLock <: AbstractSpinLock` is defined. It has extra padding to avoid false sharing (#55944).
- On Apple Silicon, `Sys.CPU_THREADS` and `Sys.EFFECTIVE_CPU_THREADS` now count all CPU cores rather than only the highest-performance tier. This affects the the following defaults: `--threads=auto` (`JULIA_NUM_THREADS=auto`), the `--gcthreads` default, `--procs=auto`, `Distributed.addprocs()`, and `JULIA_NUM_PRECOMPILE_TASKS`. Set `JULIA_CPU_THREADS` to override the detected count. The performance-core heuristic has been improved and now lives in LinearAlgebra, where it continues to size the default BLAS thread pool (#62891, JuliaLang/LinearAlgebra.jl#1686).

112.5 New library functions

- `Base.@acquire` macro for a non-closure version of `Base.acquire(f, s::Base.Semaphore)`, like `@lock` (#56845).
- `nth` function to access the n-th element of a generic iterable (#56580).
- `ispositive(::Real)` and `isnegative(::Real)` are provided for performance and convenience (#53677).
- The `fieldindex` function (to get the index of a struct's field) is now exported (#58119).
- `Base.donotdelete` is now public. It prevents dead code elimination of its arguments (#55774).
- `Sys.sysimage_target()` returns the CPU target string used to build the current system image (#58970).
- `Iterators.findeach` is a lazy version of `findall` (#54124).

112.6 New library features

- `fieldoffset` now also accepts the field name as a symbol as `fieldtype` already did (#58100).
- `sort(keys(::Dict))` and `sort(values(::Dict))` now automatically collect; they previously threw (#56978).
- `Base.AbstractOneTo` is added as a supertype of one-based axes, with `Base.OneTo` as its subtype (#56902).
- `takestring!(::IOBuffer)` removes the content from the buffer, returning the content as a `String`.

- `chopprefix` and `chopsuffix` can now also accept an `AbstractChar` as the prefix/suffix to remove.
- The `macroexpand` (with default true) and the new `macroexpand!` (with default false) functions now support a `legacyscope` boolean keyword argument to control whether to run the legacy scope resolution pass over the result. The legacy scope resolution code has known design bugs and will be disabled by default in a future version. Users should migrate now by calling `legacyscope=false` or using `macroexpand!`. This may often require fixes to the code calling `macroexpand` with `Meta.unescape` and `Meta.reescape` or by updating tests to expect `hygienic-scope` or `escape` markers might appear in the result.
- `Base.ScopedValues.LazyScopedValue{T}` is introduced for scoped values that compute their default using a `OncePerProcess{T}` callback, allowing for lazy initialization of the default value. `AbstractScopedValue` is now the abstract base type for both `ScopedValue` and `LazyScopedValue` (#59372).
- New `Base.active_manifest()` function to return the path of the active manifest, like `Base.active_project()`. Also can return the manifest that would be used for a given project file (#57937).

112.7 Standard library changes

- `mod(x::AbstractFloat, -Inf)` now returns x (as long as x is finite). This aligns with the C standard and is considered a bug fix (#47102).
- Indexless `getindex` and `setindex!` (i.e. `A[]`) on `ReinterpretArray` now correctly throw a `BoundsError` when there is more than one element (#58814).
- `randperm!` and `randcycle!` now support non-`Array` `AbstractArray` inputs, assuming they are mutable and their indices are one-based (#58596).
- `shuffle` now accepts `NTuple` arguments (#56906).

REPL

- The Julia REPL now supports bracketed paste on Windows, which should significantly speed up pasting large code blocks into the REPL (#59825).
- The REPL now provides syntax highlighting for input as you type. See the REPL docs for more info about customization.
- The REPL now supports automatic insertion of closing brackets, parentheses, and quotes. See the REPL docs for more info about customization.
- History searching has been rewritten to use a new interactive modal dialogue, using a fzf-like style.
- The display of `AbstractChars` in the main REPL mode now includes LaTeX input information like what is shown in help mode (#58181).
- Display of repeated frames and cycles in stack traces has been improved by bracketing them in the trace and treating them consistently (#55841).
- The superscript character U+107A5 𐞥 (MODIFIER LETTER SMALL Q), which was already supported in the language, can now be accessed at the REPL with \^q (#59544).

Test

- `Test` now supports the `JULIA_TEST_VERBOSE` environment variable. When set to `true`, it enables verbose testset entry/exit messages with timing information and sets the default `verbose=true` for `DefaultTestSet` to show detailed hierarchical test summaries (#59295).
- Test failures when using the `@test` macro now show evaluated arguments for all function calls (#57825, #57839).
- Transparent test sets (`@testset let`) now show context when tests error (#58727).
- `@test_throws` now supports a three-argument form `@test_throws ExceptionType pattern expr` to test both exception type and message pattern in one call (#59117).
- The testset stack was changed to use `ScopedValue` rather than task local storage (#53462).

InteractiveUtils

- Introspection utilities such as `@code_typed`, `@which` and `@edit` now accept type annotations as substitutes for values, recognizing forms such as `f(1, ::Float64, 3)` or even `sum(::Vector{T}; init = ::T) where {T<:Real}`. Type-annotated variables as in `f(val::Int; kw::Float64)` are not evaluated if the type annotation provides the necessary information, making this syntax compatible with signatures found in stacktraces (#57909, #58222).
- Code introspection macros such as `@code_lowered` and `@code_typed` now have a much better support for broadcasting expressions, including broadcasting assignments of the form `x .+= f(y)` (#58349).

Dates

- `isoweekdate`, `isoyear`, `weeksinyear` are now implemented and exported for week based calendars, following ISO week date (#48507).

112.8 External dependencies

- 7-Zip updated from p7zip v17.06 to upstream 7-Zip v25.01. On Windows, the full 7z.exe/7z.dll bundle is replaced with standalone 7za.exe, which supports fewer formats but unifies cross-platform behavior (#60025).

112.9 Deprecated or removed

- The method `merge(combine::Callable, d::AbstractDict...)` is now deprecated to favor `mergewith` instead (#59775).