Learning Rust

Personal Summary

Patrick Bucher

2024-03-14

Contents

This document is a personal summary of the book *The Rust Programming Language* by Steve Klabnik and Carol Nichols (No Starch Press, 2018). The structure of the original book has been kept on the chapter level, but may vary within the chapters. [The summary is currently extended and updated to the second edition of said book and to the 2021 edition of Rust.]

Some of the example code I made up on my own, the rest I took from the book. The formatting tool `rustfmt` has been applied for most of the example code, so the code looks different in the book and in my notes on some places.

1 Getting Started

1.1 Setup

Setup using `rustup` (make sure to have a C linker and compiler installed):

```
$ curl https://sh.rustup.rs -sSf | sh
```

Proceed with default options to get the latest stable release.

To update the environment variables, either log out and in again, or update them manually:

```
$ source $HOME/.cargo/env
```

Check the version of the Rust compiler (`rustc`) and documentation (`rustdoc`):

```
$ rustc --version
rustc 1.76.0
$ rustdoc --version
rustdoc 1.76.0
```

Open the local documentation in a browser:

```
$ rustup doc
```

Update Rust once in a while:

```
$ rustup update
```

Uninstall Rust and `rustup` when no longer needed:

```
$ rustup self uninstall
```

1.2 Hello World (manually)

Create a file `hello.rs`:

```rust
fn main() {
    println!("Hello, world!");
}
```

The following tokens are used:

- `fn` is the keyword to create a function.
- `main` is the name of the function that gets executed first when the program is started.
- `()` is an empty parameter list, because `main` doesn't expect any parameters.
- `{` starts the function body.
- `println!` is a macro (not a function) that prints a line of text to the standard output.
- `"Hello, world!"` is the string literal to be printed.
- `;` is needed at the end of every statement.
- `}` ends the function body.

Compile and run the program:

```
$ rustc hello.rs
$ ./hello
Hello, world!
```

1.3 Hello World (using Cargo)

Check if `cargo` has been installed properly:

```
$ cargo --version
1.76.0
```

Create a new binary project using `cargo` (as opposed to a library, which would take the parameter `--lib` instead of `--bin`):

```
$ cargo new hello_world --bin
```

A directory `hello_world` has been created with the following contents:

- `src/`: the folder containing the source code for the project
- `src/main.rs`: the source code file containing the `main` function
- `.git` and `.gitignore`: files for Git (define another version control system using the `--vcs` parameter)

- `Cargo.toml`: the file containing the project configuration:

```toml
[package]
name = "hello_world"
version = "0.1.0"
edition = "2021"

[dependencies]
```

TOML stands for *Tom's Obvious, Minimal Language*. The project doesn't have any dependencies yet.

Build and execute (for testing):

```
$ cargo build
$ ./target/debug/hello_world
Hello, world!
```

Build and execute (for release with optimized binary):

```
$ cargo build --release
$ ./target/release/hello_world
Hello, world!
```

A new file `Cargo.lock` is created to keep track of the dependencies versions.

Build and run in one step:

```
$ cargo run
Hello, world!
```

Only check the source code without creating a binary (faster):

```
$ cargo check
```

2 Guessing Game

2.1 User Input

Bring the standard input/output library into scope:

```
use std::io;
```

Create a new (mutable) `String` and bind it to a mutable variable:

```
let mut s = String::new();
```

Read a line of user input into a `String`, fail with an error message if it didn't work; print the input otherwise:

```
let mut input = String::new();
io::stdin()
    .read_line(&mut input)
    .expect("reading string failed"); // stops program execution
println!("{}", input);
```

Convert a the `String` from above into a number (unsigned 32-bit integer), fail with an error message if it didn't work; print the converted number otherwise:

```
let input: u32 = guess.trim().parse().expect("not a number");
println!("{}", input);
```

The second variable `input` *shadows* the first variable with the same name.

2.2 Random Numbers

In order to generate random numbers, the `rand` crate needs to be added as a dependency in `Cargo.toml`:

```
[dependencies]
rand = "0.8.5"
```

The version indicator `"0.8.5"` is shorthand for `"^0.8.5"` and means: any version with a public API compatible to version `0.8.5`.

As soon as the project is built (using `cargo build`), all the dependencies are resolved (`libc` as a dependency from `rand`, for example), and the working version configuration is written to a file `Cargo.lock`. This version configuration is used for the next build, unless the dependencies in the `Cargo.toml` file are updated, or the command `cargo update` is executed. The latter option will update the `rand` dependency. `Cargo.lock` shall be put under version control, which allows for reproducible builds.

Build and view the documentation of all the project's dependencies, including the `rand` crate, in the web browser:

```
$ cargo doc --open
```

Import the `Rng` trait, which is needed to create random numbers within a certain range:

```
use rand::Rng;
```

Create a random number within a range (here, both the lower and upper bound are inclusive):

```
let min = 1;
let max = 101;
let number = rand::thread_rng().gen_range(1..=100);
println!("Random number 1..100: {}", number);
```

2.3 Compare Numbers

Make the `Ordering` enum available, which covers all the possible results of a comparison:

```
use std::cmp::Ordering;
```

Compare two numbers which one another using pattern matching:

```
let a = 3;
let b = 5;
match a.cmp(&b) {
    Ordering::Less => println!("a<b"),
    Ordering::Greater => println!("a>b"),
    Ordering::Equal => println!("a=b"),
}
```

2.4 Loops and User Input

Request user input until a number is entered using a infinite loop:

```
loop {
    let mut input = String::new();
    println!("enter a number");
    match io::stdin().read_line(&mut input) {
        Result::Ok(_) => (), // do nothing
        Result::Err(_) => continue, // once again
    }
    let input: u32 = match input.trim().parse() {
        Ok(num) => num, // parsed input as the match expression's result
        Err(_) => {
            println!("not a number");
            continue; // once again
        }
    };
    // do something sensible with the number entered
    break;
}
```

3 Common Programming Concepts

Rust offers the common features of structured programming languages, but implements some in a special way.

3.1 Variables and Constants

Variables are immutable by default:

```
let x = 3;
x = 5; // error: cannot assign twice to immutable variable `x`
```

Only the values of variables declared as mutable can be changed:

```
let mut x = 3;
x = 5; // OK
```

Variables can be redeclared, even their type can be changed:

```
let x = 3;
let x = x * x;
let x = x + 1;
println!("{}", x); // 10

let x = "10110";
let x = x.len();
println!("{}", x); // 5
```

In the examples above, five immutable variables called x have been declared; no variable was ever changed. This technique is called *shadowing*: the second x shadows the first x, the third x shadows the second x, etc.

However, an immutable variable cannot be shadowed by a mutable variable.

Constants are a lot like immutable variables, but:

1. cannot be declared as mutable,
2. are declared using the `const` keyword (as apposed to `let`),
3. can be declared in any scope,
4. can only be assigned to expressions known at compile time,
5. use the `ALL_UPPERCASE` naming convention, and
6. require a type annotation.

```
const SPEED_OF_LIGHT: u32 = 299_792_458; // in meters per second (vacuum)
```

The _ (underscore) is for optical groupings of three and has no special meaning.

3.2 Data Types

Rust is statically typed. The types of variables must be known at compile time. In many cases, the type can be inferred from the context, in other cases, the type must be annotated:

```
let foo = "test"; // string inferred
let bar = 10_000; // integer inferred

let qux: u32 = "42".parse().expect("can't parse to u32");
```

Without the annotation for `qux`, the compiler wouldn't know to which type `parse()` has to convert the given string.

3.2.1 Primitive Types

Rust offers four basic types of scalar (single values): integers, floating point numbers, booleans and characters.

3.2.1.1 Integers Integers exist as signed and unsigned variants:

Size	Signed	Unsigned
8-bit	`i8`	`u8`
16-bit	`i16`	`u16`
32-bit	`i32`	`u32`
64-bit	`i64`	`u64`
Architecture	`isize`	`usize`

In general, `i32` works fastest, even on 64-bit platforms.

The type names can be used as suffixes in literals:

```
let a: u8 = 255;
let a = 255u8;

let b: i8 = -128;
let b = -128i8;
```

Integer literals can be written in binary, octal, decimal (default) and hexadecimal notation:

```
let base2 = 0b01100100; // binary: prefix 0b
let base8 = 0755; // octal: prefix 0
let base10 = 1234567890; // decimal: no prefix
let base16: u32 = 0xdeadbeef; // hexadecimal: prefix 0x
```

The hexadecimal number needs a type annotation, because the signed 32-bit integer (`i32`) inferred is too small for it.

There is a special byte prefix to convert ASCII characters into numbers:

```
let ascii_capital_a = b'A'; // 65
```

Arithmetic operators can be applied both to variables and literals:

```
let sum = 3 + 5; // 8
let difference = sum - 5 // 3
let product = difference * sum; // 24
let quotient = product / 2; // 12
let remainder = quotient * 5; // 2
```

3.2.1.2 Floating Point Numbers

Rust supports floating point numbers according to the IEEE-754 standard with single precision (`f32`) and double precision (`f64`). A lot of the integer notations and conventions can be used for floating point numbers, too:

```
let a = 13.2;
let b: f32 = 3.41;
let c = 5.324f64;
let d = 123_456.789;
```

3.2.1.3 Boolean

The `bool` type knows two values: `true` and `false`:

```
let right = true; // type inferred
let wrong: bool = false; // with type annotation
```

3.2.1.4 Character

Characters in Rust are UTF-8 encoded, and therefore not the same thing as a single byte:

```rust
let latin_lower_c = 'c'; // 99 (requires one byte)
let cyrillic_upper_d = 'Д'; // 1044 (requires two bytes)
```

3.2.2 Compound Types

Rust has two compound types: tuples and arrays.

3.2.2.1 Tuples

A tuple groups together values of (possibly) different types. A tuple's elements can be accessed using dot-notation (using a zero-based index) or through the means of destructuring:

```rust
let t: (i32, f64, u8) = (123, 4.56, 78);

// dot-notation
let a = t.0;
let b = t.1;
let c = t.2;

// destructuring
let (a, b, c) = t;
```

3.2.2.2 Arrays

Rust's arrays have a fixed size, so elements can be neither added nor removed, but replaced if the array is declared as mutable. Unlike tupels, all elements of an array must be of the same type.

```rust
let mut a = [1, 2, 3, 4, 5];
a[0] = 5;
a[4] = 1;
println!("[{},{},{},{},{}]", a[0], a[1], a[2], a[3], a[4]); // [5,2,3,4,1]
```

Both element type and number of elements can be annotated:

```rust
let nums: [u32; 3] = [1, 2, 3];
```

The usage of out-of-bounds indices either causes a runtime panic (if the index value is computed at runtime) or doesn't even compile (if the index value can be computed at compile time).

3.3 Functions

Function names should follow the snake_case convention, i.e. all letters are in lowercase, and the words are separated by an underscore _.

The order of the function's declarations doesn't matter; function calls are possible forwards and backwards.

The parameter and return types of a function are not inferred and must be declared explicitly. Like for variables, the parameter types are annotated; the return type is indicated after an arrow (->):

```
fn sum_up(a: i32, b: i32) -> i32 {
    a + b
}
```

A function can either end in an expression (as above), which is used as the function's return value, or an expression can be returned explicitly using the return statement:

```
fn sum_up(a: i32, b: i32) -> i32 {
    return a + b;
}
```

Any block can return a value:

```
let y = {
    let x = 3;
    x + 5 // expression: no semicolon
}; // semicolon, ending the let statement
```

Because statements do *not* return values, this code doesn't compile:

```
let c = (let b = (let c = 1)); // error: expected expression
```

3.4 Comments

Single-line comments start with // and end at the line's end.

Multi-line comments start with /* and end with */.

3.5 Control Flow

3.5.1 Conditional Execution

Only `bool` expressions are allowed for `if` conditions:

```
if x < 4 {
    println!("low");
} else if x < 7 {
    println!("medium");
} else {
    println!("high");
}
```

`if` is an expression, not a statement, and therefore can return a value:

```
let max = if a > b {
    a
} else {
    b
};
```

It's important that the expressions of both arms are of the same data type!

3.5.2 Loops

`loop` runs infinitely, unless ended with `break`:

```
let numbers = [1, 2, 3, 4, 5];
let mut i = 0;
loop {
    println!("{}", numbers[i]);
    if i >= 4 {
        break;
    } else {
        i += 1;
    }
}
```

`while` checks a condition on every iteration before the its block is executed:

```
let numbers = [1, 2, 3, 4, 5];
while i < 5 {
    println!("{}", numbers[i]);
    i += 1;
}
```

`for` iterates over the items of a collection, e.g. an array:

```
let numbers = [1, 2, 3, 4, 5];
for i in numbers.iter() {
    println!("{}", i);
}
```

The `for` loop is by far the most commonly used in Rust.

`continue` leaves the loop's block and moved forward to the next iteration:

```
let numbers = [1, 2, 3, 4, 5];
for i in numbers.iter() {
    if i % 2 == 0 {
        continue; // skip even numbers
    }
    println!("{}", i);
}
```

Loops can return values and, thus, be used as expressions:

```
fn double(x: u32, threshold: u32) -> u32 {
    let mut temp = x;
    let mut i = 0;
    let times = loop {
        temp *= 2;
        i += 1;
        if temp > threshold {
            break i;
        }
    };
    times
}
```

The above function `double` doubles the given value `x` until `threshold` is exceeded, after which the number of performed operations is returned—both from the loop, and from the function.

4 Ownership

Rust neither has garbage collection nor requires manual memory management. It uses a third approach: Memory is managed through a system of rules enforced by the compiler—the ownership model. The rules are:

1. Each value has a variable that's called its *owner*.
2. There can be only one owner at a time.
3. When the owner goes out of scope, the value is dropped.

Rust calls the `drop()` function for every variable at the end of its scope to free the memory. The owner variable will be no longer valid from this point.

4.1 Stack and Heap

Variables of primitive types (integers, floats, booleans, characters and tuples solely consisting of primitive types) are stored entirely on the stack. If a variable is assigned to a variable of a primitive type, the whole stack content is copied:

```
let a = 3;
let b = a; // the value of a is copied
```

Types with sizes unknown at compile time, like strings, are stored on the heap:

```
let s1 = String::from("hello");
let s2 = s1; // the value of s1 is NOT copied
```

Copying the content of `s1` would be expensive in terms of runtime performance. Therefore, only the pointer to the string's heap memory location is copied, i.e. the variable's stack content is (just like for primitives), but here the stack content is a pointer to the heap. (Note: This is *not* what actually happens; see below!)

The content of a heap variable can be created using the `clone()` method:

```
let s1 = String::from("hello");
let s2 = s1.clone();
```

4.2 Moving

The different handling of stack and heap objects has consequences in regard to memory management. The automatic `drop()` on primitive variables at the end of a scope is unproblematic:

```
// before: a and b out of scope
{
    let a = 3;
    let b = a;
}
// after: a and b out of scope
```

The value of `a` was copied, and only the stack memory of `a` and `b` is freed. The behaviour is different for objects on the heap:

```
// before: s1 and s2 out of scope
{
    let s1 = String::from("hello");
    let s2 = s1;
    // call drop() on s1 and s2
}
// after: s1 and s2 out of scope
```

Freeing the same memory twice corrupts the memory, could cause a program to crash and might cause a security vulnerability. To prevent such problems, the assignment `s2 = s1` in the program above does *not* create a *shallow copy* (and also not a *deep copy*, as already mentioned). The value is instead *moved* from `s1` to `s2`, and `s2` becomes the new owner. The variable `s1` cannot be used any longer from that point:

```
let s1 = String::from("hello");
let s2 = s1; // value of s1 moves to s2
println!("s1={}", s1); // invalid, value moved to s2
println!("s2={}", s2); // valid, s2 is the new owner
```

4.2.1 Move on Function Call

A function call behaves a lot like an assignment in terms of ownership of the parameters. A function expecting a `String` will take ownership of that value, but a integer value will just be copied:

```
fn move_or_copy(str: String, nbr: i32) {
    println!("str={}, nbr={}", str, nbr);
}

fn main() {
    let s = String::from("abc");
    let i = 42;
    move_or_copy(s, i); // move s, copy i
    println!("s={}", s); // invalid: s was moved
    println!("i={}", i); // valid: i was copied
}
```

The ownership of s can be moved back by returning it from the function:

```
fn move_or_copy(str: String, nbr: i32) -> String {
    println!("str={}, nbr={}", str, nbr);
    str // return
}

fn main() {
    let s = String::from("abc");
    let i = 42;
    let s2 = move_or_copy(s, i); // take the ownership of the object back
    println!("s2={}", s2); // valid now!
    println!("i={}", i);
}
```

4.3 Borrowing with References

Obtaining the ownership of a variable and giving it back is tedious. A variable can be *borrowed* instead by using a reference. A reference is denoted by an ampersand in front of the type (`&String`) and value (`&s`):

```
fn borrow(str: &String) { // &String: reference to String
    println!("str={}", str);
}

fn main() {
    let s = String::from("abc");
    borrow(&s); // &s is a reference to s
    println!("s={}", s); // valid: s was only borrowed, not owned
}
```

The value of a reference can only be modified if the reference is mutable. Mutable references are denoted by the token `&mut` in front of the type (`&mut String`) and value (`&mut s`). Mutable references can only be acquired from mutable variables:

```
fn manipulate(str: &mut String) { // &mut String: mutable reference to String
    str.push_str("...xyz"); // allows for manipulation
}

fn main() {
    let mut s = String::from("abc"); // mutable variable
    manipulate(&mut s); // mutable reference
    println!("s={}", s); // valid: s was only borrowed, not owned
}
```

4.3.1 Data Races

Having multiple references to a memory object allows for *data races*: The values is updated through one reference, and the other references are not aware of that. Rust eliminates data races by enforcing a very strict set of rules:

> One can only have:

1) either one mutable reference
2) or multiple immutable references

to the same value in the same scope.

One mutable reference is ok:

```
let mut s = String::from("hello");
let r1 = &mut s;
println!("{}", r1);
```

Two mutable references are *not* ok:

```
let mut s = String::from("hello");
let r1 = &mut s;
let r2 = &mut s; // error, only one mutable reference allowed
println!("{}", r1);
println!("{}", r2);
```

However, two mutable references *in different scopes* are ok:

```
let mut s = String::from("hello");
{
    let r1 = &mut s;
    println!("{}", r1);
} // r1 goes out of scope
let r2 = &mut s;
println!("{}", r2);
```

Multiple immutable references are ok:

```
let mut s = String::from("hello");
let r1 = &s;
let r2 = &s;
println!("{}", r1);
println!("{}", r2);
```

But only as long as there is no mutable reference:

```
let mut s = String::from("hello");
let r1 = &s;
let r2 = &s;
let r3 = &mut s; // error, one mutable or multiple immutable references allowed
println!("{}", r1);
println!("{}", r2);
println!("{}", r3);
```

4.3.2 Dangling Pointers

A function returning a pointer to a heap value that goes out of scope at the end of that function is called a *dangling pointer*. Using dangling pointers can crash the program and cause severe security problems. Rust doesn't allow dangling pointers:

```
fn dangle() -> &String {
    let s = String::from("hello");
    &s // return a reference to an object owned by this function
} // s goes out of scope

fn main() {
    println!("{}", dangle()); // invalid: dangling ponter
}
```

The function must hand over the ownership of the objects it created for further use:

```
fn dangle() -> String { // return the string (with ownership)
    let s = String::from("hello");
    s // move
}

fn main() {
    println!("{}", dangle()); // valid: owned value
}
```

4.4 String Slices

A string slice is a reference to a part of a string. A slice stores the starting point within the string and its length. The slice boundaries are stated with a inclusive lower and a exclusive upper bound, so that `upper-lower` computes to the slice length:

```
let s = String::from("hello world");
let hello = &s[0..5];
let world = &s[6..11];
println!("{}, {}!", hello, world); // hello, world!
```

The lower and upper bound can be omitted, defaulting to zero resp. to the string length. If both bounds are omitted, the slice spans over the whole string:

```
let s = String::from("hello world");
let hello = &s[..5]; // lower bound defaults to 0
let world = &s[6..]; // upper bound defaults to s.len()
println!("{}, {}!", hello, world); // hello, world!
```

```
let slice = &s[..]; // no bounds: span over whole string
println!("{}", slice); // hello world
```

When using multi-byte/non-ASCII characters, the slice must neither start nor end in between the UTF-8 character boundaries:

```
let privet = String::from("привет");
let pri = &privet[..6]; // ok: "при"
let vet = &privet[6..]; // ok: "вет"
println!("{}{}", pri, vet);
let pri_ = &privet[..7]; // wrong: "при" + first byte of 'в'
```

A string cannot be modified when a slice is referring to it:

```
let mut s = String::from("abcdefg");
let abc = &s[..4];
s.push_str("hijklmnop"); // invalid: already borrowed immutably
```

Moving the immutable reference into its own scope solves the problem:

```
let mut s = String::from("abcdefg");
{
    let abc = &s[..4];
}
s.push_str("hijklmnop"); // valid: immutable borrow already dropped
```

String literals are string slices referring to a certain memory area within the binary program. It's a good practice to accept string slices (type `&str`) as function parameters, because string literals already are string slices, and instances of `String` are cast to a string slice by referring:

```
fn quote(s: &str) {
    println!("«{}»", s);
}

fn main() {
    let s = String::from("Hello, World!");
    quote(&s); // String object: cast by referring
    quote("Hello, World!"); // string literal: already a slice
}
```

Slices can also be created on other sequences, such as integer arrays:

```
let fib = [1, 1, 2, 3, 5, 8, 13, 21];
let slice = &fib[2..7]; // [2, 3, 5, 8, 13]
```

5 Structs

Structs name and group together multiple related values of possibly different types to a new type. Unlike tuples, structs give names to the whole and its parts, and the order of fields neither matters for initialization nor accessing values.

A struct is defined using the keyword struct, a name (in capitals) and a list of fields (name-type pairs) in curly braces:

```
struct Employee {
    name: String,
    position: String,
    logins: u64,
    active: bool,
}
```

A new instance of a struct is created by defining its field in key-value notation. All fields are mandatory and need to be initialized:

```
let dilbert = Employee {
    name: String::from("Dilbert"),
    position: String::from("Engineer"),
    logins: 962,
    active: true,
};
```

When marked as mutable, fields of a struct instance can be changed:

```
let mut dilbert = Employee {
    name: String::from("Dilbert"),
    position: String::from("Engineer"),
    logins: 962,
    active: true,
};
dilbert.name = String::from("Dilberto");
dilbert.position = String::from("Head of Engineering");
dilbert.logins += 1;
dilbert.active = false;
```

Only the struct instance as a whole can be marked as mutable, not single fields.

A struct without any fields is called a *unit-like* struct. It can be useful where a value is needed formally, but doesn't matter:

```
struct Empty{};
let container = Empty{};
```

Factory functions are helpful to create new struct instances, especially if they consist of a mix of custom and default values:

```rust
fn create_employee(name: String, position: String) -> Employee {
    Employee {
        name: name,
        position: position,
        logins: 0,
        active: true,
    }
}
```

If the field and the variable assigned to it have the same name, the *field init shorthand* syntax allows to only indicate the name once:

```rust
fn create_employee(name: String, position: String) -> Employee {
    Employee {
        name,
        position,
        logins: 0,
        active: true,
    }
}
```

This not only works for function parameters, but for any variables in scope:

```rust
let name = String::from("Catbert");
let position = String::from("Evil Genius");
let catbert = Employee {
    name,
    position,
    logins: 0,
    active: true,
};
```

New instances can be created based on existing ones:

```rust
let wally = Employee {
    name: String::from("Wally"),
    position: dilbert.position,
    logins: 312,
    active: dilbert.active,
};
```

The *struct update* syntax allows to initialize the remaining fields based on the value of an existing instance:

```
let wally = Employee {
    name: String::from("Wally"),
    logins: 312,
    ..dilbert // use Dilbert's values for fields not yet initialized
};
```

5.1 Tuple Structs

A tuple struct is a tuple with a name. The name belongs to the type definition, therefore this two tuple structs are not compatible to each other:

```
struct RGB(u8, u8, u8);
struct Block(u8, u8, u8);

let red = RGB(255, 0, 0);
let cube = Block(1, 1, 1);
```

A loose bunch of variables on one side and a struct on the other side can be seen as the two extremes of a continuum:

1. *loose variables* can be grouped together to a *tuple*
2. a *tuple* can be named as a whole to get a *named tuple*
3. the fields of a *named tuple* can be named to get a *struct*

5.2 Debug Output

Struct instances can be printed out if they derive the trait `std::fmt::Debug`:

```
#[derive(Debug)]
struct Employee {
    name: String,
    position: String,
    logins: u64,
    active: bool,
}

fn main() {
    let dilbert = Employee {
        name: String::from("Dilbert"),
        position: String::from("Engineer"),
        logins: 962,
        active: true,
    };
    println!("{:?}", dilbert);
    println!("{:#?}", dilbert);
}
```

The `{:?}` output format prints the struct on a single line, whereas the `{:#?}` output format uses multiple lines and indents the fields:

```
Employee { name: "Dilbert", position: "Engineer", logins: 962, active: true }
Employee {
    name: "Dilbert",
    position: "Engineer",
    logins: 962,
    active: true
}
```

5.3 Methods

A method is a function defined in the context of a struct. The instance of the struct the method is called on is automatically provided as the first parameter called `self`. The methods of a struct must be declared within one or multiple `impl` blocks:

```
struct Rectangle {
    width: u32,
    height: u32,
}

impl Rectangle {
    fn area(&self) -> u32 {
        self.width * self.height
    }
}
```

The `area` method accepts a reference to a `Rectangle` (the type doesn't need to be declared, because it can be inferred). The method can be called using dot notation on the struct instance (the reference operator `&` is optional):

```
fn main() {
    let r = Rectangle {
        width: 3,
        height: 4,
    };
    println!("{}", r.area()); // 12
    println!("{}", &r.area()); // with optional reference operator
}
```

The `self` instance can be modified by accepting a mutable reference:

```rust
impl Rectangle {
    fn stretch(&mut self, factor: u32) {
        self.width *= factor;
        self.height *= factor;
    }
}

fn main() {
    let mut r = Rectangle {
        width: 3,
        height: 4,
    };
    println!("{}", r.area()); // 3 * 4 = 12
    r.stretch(2);
    println!("{}", r.area()); // 6 * 8 = 48
}
```

Methods that own the `self` parameter are rare. They're helpful when a new instance is created based on an old one, and the old instance must no longer be used afterwards. The old instance is *consumed* by the method:

```rust
impl Rectangle {
    fn transform(self, factor: u32) -> Rectangle {
        Rectangle {
            width: self.width * factor,
            height: self.height * factor,
        }
    }
}

fn main() {
    let r = Rectangle {
        width: 3,
        height: 4,
    };
    let r = r.transform(2);
    println!("{}", r.area());
}
```

Associated functions, similar to `static` methods in other programming languages, belong to a struct, but do not take a `self` parameter; they are often used as constructors:

```
impl Rectangle {
    fn square(size: u32) -> Rectangle {
        Rectangle {
            width: size,
            height: size,
        }
    }
}
```

An associated function can be called using double colon notation on the struct, as already used for `String::from()` before:

```
fn main() {
    let r = Rectangle::square(3);
    println!("w: {}, h: {}", r.width, r.height); // w: 3, h: 3
}
```

6 Enumerations and Pattern Matching

Enumerations (short: *enums*) are types defined by enumerating its possible values and encode meaning along with data. The enum's different values are called its *variants*.

A enum is defined using the `enum` keyword and a list of variants within curly braces:

```
enum ColorType {
    RGBA,
    CMYK,
}
```

The variants of an enum belong to the same type. They are namespaced under the enum's name and need to be qualified in order to be used:

```
let rgba = ColorType::RGBA;
let cmyk = ColorType::CMYK;
```

A function can accept values of either variant by annotating the enum's name as the expected type:

```
fn list_colors(color_type: ColorType) {
    // ...
}

fn main() {
    list_colors(ColorType::RGBA;);
    list_colors(ColorType::CMYK);
}
```

Variants can hold their own data. The amount, structure and type of the associated data can vary between different variants. It is possible to use single scalars, tuples, anonymous structs and even other enums:

```
#[derive(Debug)]
enum ColorType {
    Unspecified,           // no data
    Named(String),         // single value
    RGBA(u8, u8, u8, f32), // tuple
    CMYK {
        cyan: u8,
        magenta: u8,
        yellow: u8,
        black: u8,
    }, // anonymous struct
}
```

Even though the variants are made up of different types, they still belong to the same enum type.

Enums can have methods attached to them in `impl` blocks:

```
impl ColorType {
    fn print(&self) {
        println!("{:?}", self);
    }
}

fn main() {
    let none = ColorType::Unspecified;
    let fuchsia = ColorType::Named(String::from("fuchsia"));
    let red = ColorType::RGBA(255, 0, 0, 0.5);
    let yellow = ColorType::CMYK {
        cyan: 0,
        magenta: 0,
        yellow: 100,
        black: 0,
    };

    none.print();
    fuchsia.print();
    red.print();
    yellow.print();
}
```

Output:

```
Unspecified
Named("fuchsia")
RGBA(255, 0, 0, 0.5)
CMYK { cyan: 0, magenta: 0, yellow: 100, black: 0 }
```

6.1 The `Option<T>` Enumeration

Unlike many other programming language, Rust has no `null` (or `nil`) references. The absence (and presence) of a value is instead expressed by the `Option<T>`; an enum with two variants and a type parameter `T`:

```
enum Option<T> {
    Some(T),
    None,
}
```

The variant `Some` holds a value of type `T`. The variant `None` is used to signify the absence of a value and therefore does not hold any. The enum `Option<T>` and its variants `Some(T)` and `None` are included in the prelude and hence are not required to be made available first. They can be used without qualification:

```
let a_number = Some(42);
let a_word = Some(String::from("whatever"));
let none: Option<i32> = None;
```

The type can be inferred in case of `Some` but must be annotated for `None`, for there is no value to infer from.

A instance of `Option<T>` can not be treated like an instance of `T`:

```
let a: i32 = 3;
let b: Option<i32> = Some(4);
let c: Option<i32> = None;

let x = a + b; // error
let y = a + c; // error
```

The variable a is of type i32, while b and c are of type Option<i32>. Trying to handle them alike causes a compilation error. This is exactly the point, because adding a *only possibly* available value to a existing value cannot be guaranteed to work. The compiler makes sure that a value is always there when needed.

Option<T> must be converted to T before it can be used. The enum offers methods for that purpose, but the general approach is to use pattern matching.

6.2 Pattern Matching

The match control flow operator allows to compare a value against a series of patterns and executes a branch of code for the matching pattern.

In the context of enums, match allows to execute different code for a value according to its variant. Every variant has an expression, statement or block assigned with the => operator, called the variant's *arm*:

```
enum Animal {
    Blobfish,
    Human,
    Fox,
    Octopus,
    Centipede,
}

fn number_of_legs(animal: Animal) -> u8 {
    match animal {
        Animal::Blobfish => 0,
        Animal::Human => {
            println!("not counting arms");
            2
        },
        Animal::Fox => 4,
        Animal::Octopus => {
            println!("arms or legs? legs!");
            return 8;
        },
        Animal::Centipede => 100,
    }
}

fn main() {
    let johnny = Animal::Human;
    let legs = number_of_legs(johnny);
    println!("{} legs", legs);
}
```

If one or many variants were missing, the program would not compile:

```
fn number_of_legs(animal: Animal) -> u8 {
    match animal {
        Animal::Blobfish => 0,
        Animal::Fox => 4,
        Animal::Centipede => 100,
    }
}
```

```
match animal {
      ^^^^^^ patterns `Human` and `Octopus` not covered
```

Matches are said to be *exhaustive*; the compiler makes sure that all the possibilities are handled. If one or many variants are *not* of interest, the _ pattern, which matches to anything, can be used for the last arm. (If it were not the last arm, the subsequent patterns could not possibly match, because _ already matched everything.)

```
fn number_of_legs(animal: Animal) -> u8 {
    match animal {
        Animal::Blobfish => 0,
        Animal::Fox => 4,
        Animal::Centipede => 100,
        _ => 0,
    }
}
```

If no code were to be executed for the default case, the unit value `()` can be used:

```
match animal {
    Animal::Blobfish => println!("0"),
    Animal::Fox => println!("4"),
    Animal::Centipede => println!("100"),
    _ => (),
}
```

This only works if `match` is used as a statement as opposed to an expression; the latter always needs to yield a value.

If only a single variant is of interest, the `if let` construct can be used to make the code more concise. Consider this `match` expression:

```
let val = Some(42);
match val {
    Some(42) => println!("correct"),
    _ => (),
}
```

Which can be rewritten using the `if let [pattern] = [value]` pattern as follows:

```
let val = Some(42);
if let Some(42) = val {
    println!("correct");
}
```

The resulting code is shorter, but the compiler no longer checks if all variants are being handled.

One common use case for pattern matching is to extract a value of an enum's variant. Consider this enum with a enhanced `Centipede` definition, of which the value needs to be extracted:

```
enum Animal {
    Blobfish,
    Human,
    Fox,
    Octopus,
    Centipede(String),
}

fn number_of_legs(animal: Animal) -> u8 {
    match animal {
        Animal::Blobfish => 0,
        Animal::Human => 2,
        Animal::Fox => 4,
        Animal::Octopus => 8,
        Animal::Centipede(kind) => {
            if kind == "Stone Centipede" {
                30
            } else if kind == "Giant Readhead Centipede" {
                42
            } else {
                100
            }
        }
    }
}

fn main() {
```

```rust
    let centi = Animal::Centipede(String::from("Stone Centipede"));
    let legs = number_of_legs(centi);
    println!("{} legs", legs);
}
```

The `Animal::Centipede(kind)` pattern is said to *bind to a value*. This technique is also used to unpack the `T` instance of a `Option<T>`. Consider this safe division function, which returns the resulting quotient wrapped in a `Option<f32>`—or `None`, if the division is undefined when the divisor is zero:

```rust
fn divide(dividend: i32, divisor: i32) -> Option<f32> {
    if divisor == 0 {
        return None;
    }
    Some(dividend as f32 / divisor as f32)
}
```

The result of the `divide()` function is handled with pattern matching, ensuring that always a value present by mapping `None` to `0`:

```rust
let a = 10;
let b = 3;
let c = 0;

let x = match divide(a, b) {
    Some(q) => q,
    None => 0.0,
};
let y = match divide(a, c) {
    Some(q) => q,
    None => 0.0,
};

println!("x={}, y={}", x, y); // x=3.3333333, y=0
```

7 Modules

Just like lines of code can be organized as functions, entities like functions, enums, structs and constants can be organized in a higher logical unit: modules.

The following examples are demonstrated using two different crates:

1. `netlib`: a library crate containing the modules
    - `src/lib.rs` as the starting point

2. `nettool`: a binary crate using the modules
    - `src/main.rs` as the starting point

The crates are created in the same directory using `cargo`:

```
$ cargo new netlib --lib
$ cargo new nettool --bin
```

Modules are not restricted to the usage within library crates. The purpose of this structure is to demonstrate how a binary crate can make use of a library, and libraries offer not only loose functions in general, but functions organized in a module structure.

A module is declared using the `mod` keyword, following the module name and the module definition (its content) within curly braces (`netlib/src/lib.rs`):

```
mod network {
    fn connect() {}
}
```

The functions of a module are said to live in the module's *namespace*, which is expressed by the reference `network::connect` for the above example.

Multiple modules can be declared alongside in the same file:

```
mod network {
    fn connect() {}
}
mod database {
    fn connect() {}
}
```

The code compiles, even though two functions called "`connect`" are declared in the same file, because the functions are in two different modules and hence in different namespaces.

7.1 Hierarchy: Logical and Physical

Modules can be nested to represent hierarchic module structures:

```
mod server {
    fn serve() {}
    mod producer {
        fn produce() {}
    }
}
mod client {
    fn consume() {}
}
mod database {
    fn backup() {}
}
```

A file containing multiple and/or nested modules tends to become big (many definitions) and hard to read (deeper indentation levels). It makes sense to not only organize modules logically, but also physically in terms of files.

A module's declaration and definition can be taken apart. In case of the module `database`, the definition remains in the file `src/lib.rs`:

```
mod server {
    mod producer {}
}
mod client {}
mod database;
```

The definition (consisting of a single function) is then expected to be in a file named after the module (`src/database.rs`):

```
mod database {
    fn backup() {}
}
```

During the compilation process, `lib.rs` (library crate) or `main.rs` (binary crate), respectively, are considered for declarations. The module declaration points the compiler forward to another file; the module `database` is looked for in the file `database.rs` in the `src/` folder.

Extracting a nested module (`server`) with one or many child modules (`producer`) of its own requires another organization:

1. The parent module `server` must reside in a subfolder named after the module `src/server/` in a file called `mod.rs`.
2. The child module `producer` must reside in the folder of its parent module `src/server/` in a file named after the child module `producer.rs`.

The root file `lib.rs`, with the definition of server extracted, looks like this:

```
mod server;
mod client {
    fn consume() {}
}
mod database;
```

The file `server/mod.rs` only contains the server module's definitions, *not the declaration line* `mod server` itself, which is already implied by the context:

```
mod producer;
fn serve() {}
```

The submodule `server::producer` must reside in the file `server/producer.rs`:

```
mod producer {
    fn produce() {}
}
```

In summary, the module structure looks like this:

```
netlib
    server
        producer
    database
```

While the file system structure looks like this:

```
src/
    lib.rs
    server/
        mod.rs
        producer.rs
    database.rs
```

7.2 Visibility: Private and Public

Functions and other items are private to the enclosing module by default, and hence cannot be accessed from the outside. The `pub` keyword makes them public, so that they can be used from the outside, too.

A public function or module is only accessible from the outside if all the enclosing modules are declared public as well:

```
mod parent {
    pub mod bazzer {
        pub mod open {
            pub fn baz() {}
        }
    }
    pub mod quxer {
        mod locked { // private module
            pub fn qux() {}
        }
    }
}

fn main() {
    parent::bazzer::open::baz(); // works: whole path is public
    parent::quxer::locked::qux(); // doesn't work: private module in path
}
```

Private items—modules, functions, etc.—can only be accessed from:

1. its immediate parent module
2. the parent's child modules (siblings)
   - the parent module can be referred to using the `super` keyword in the module path

```
mod parent {
    mod first { // private module
        pub fn one() {} // public function
    }
    fn zero() {
        first::one(); // 1. immediate parent module
    }
    mod second {
        fn two() {
            super::first::one(); // 2. parent's child module (sibling)
        }
    }
}
```

Private functions that aren't used within their visible context cause a compiler warning, because they aren't needed and therefore be better deleted. Public functions—whether used locally or not—do not cause such a warning, because they are intended for external use; and the external context is not known during compilation. (A library doesn't know from where and how it is going to be used.)

7.3 Using External Crates and Modules

Library crates are only useful when they are eventually used by some binary. Let's assume the following crates are created within the same parent directory, a `house` (library) serving its `resident` (binary):

```
$ cargo new house --lib
$ cargo new resident --bin
```

The library consists of a module with multiple submodules in `house/src/lib.rs`:

```
pub mod basement {
    pub mod freezer {
        pub mod door {
            pub fn open() {
                println!("open the door of the freezer in the basement");
            }
            pub fn close() {
                println!("close the door of the freezer in the basement");
            }
        }
    }
}
```

If the resident binary wants to make use of the house library, it must declare that dependency in resident/Cargo.toml:

```
[package]
name = "resident"
version = "0.1.0"
authors = ["patrickbucher <patrick.bucher@stud.hslu.ch>"]
edition = "2018"

[dependencies]
house = { path = "../house", version = "*" }
```

The path is indicated relatively (../house), and any version can be used (*).

In order to use the external house crate from the library, it must also be declared in the code (resident/src/main.rs):

```
extern crate house;

fn main() {
    house::basement::freezer::door::open();
    house::basement::freezer::door::close();
}
```

Libraries with a deep module hierarchy are painful to use if every path had to be qualified absolutely. The use keyword brings a module's content into scope, so that its content can be used without further qualification.

```
extern crate house;

use house::basement::freezer::door;

fn main() {
    door::open();
    door::close();
}
```

The use keyword can be applied at any level. Consider this additional enum defined in `house/src/lib.rs`:

```
pub enum FrozenFoodType {
    IceCream,
    Meat,
    Vegetables,
    IceRocks,
}
```

It's possible to import just the enum:

```
extern crate house;

use house::FrozenFoodType;

fn main() {
    let icecream = FrozenFoodType::IceCream;
}
```

Or, one level deeper, directly the variants:

```
use house::FrozenFoodType::IceCream;
```

Multiple variants (or items in general) can also be imported as a list:

```
use house::FrozenFoodType::{IceCream, Meat};
```

It's also possible to import *all* items of an entity, which should be used sparingly in order to not pollute the namespace:

```
use house::FrozenFoodType::*;
```

8 Common Collections

Rust has different kinds of collections, which allow to store multiple elements of the same type. The elements are stored on the heap, and therefore, unlike arrays, the number of items is flexible.

8.1 Vector

The type `Vec<T>` describes a *vector*, which stores a list of items in a continuous memory area. A vector can be created using its associated `new()` function:

```
let v: Vec<i32> = Vec::new();
```

A type annotation (within angle brackets) is needed, because there are no elements yet to infer the type from. If a vector is to be created from existing items, the `vec!` macro (included in the prelude) is more convenient:

```
let v = vec![1, 2, 3];
```

Elements can be added to a mutable vector using the `push()` method:

```
let mut v = vec![1, 2, 3];
v.push(4);
v.push(5);
v.push(6);
```

There are two ways of reading the elements of a vector `v`:

1. Using *indexing syntax* `&v[i]`, which panics if an invalid index is used.
2. Using the `v.get(i)` method, which returns an `Option<T>` with either a value `Some(T)` for valid indices or the `None` variant for invalid indices.

```
let v = vec![1, 2, 3];
let first = &v[0];     // 1: i32
let second = v.get(1); // Some(2): Option<i32>
let fourth = &v[3];    // panic!
let fifth = v.get(4);  // None: Option<i32>
```

Storing a reference of a vector item is borrowing from the vector. As long as a borrow is held (and used), the vector cannot be modified by adding items to it:

```
let mut v = vec![1, 2, 3];
let one = &v[0];      // immutable borrow
v.push(4);            // error: cannot borrow as mutable
println!("{}", one); // use immutable borrow later
```

The reason for this restriction is the way the memory of a vector is organized: If the vector no longer fits into its current memory area after an additional item is added, the *whole* vector needs to be moved to a bigger continuous memory area, and thus rendering the existing references obsolete.

The items of a vector can be iterated over using the `for`/`in` loop:

```
let v = vec![1, 2, 3];
for i in v {
    println!("{}", i);
}
```

The items of a mutable vector can be modified in an iteration, if the vector is borrowed mutably for the operation, and the item is dereferenced upon modification:

```
let mut v = vec![1, 2, 3];
for i in &mut v {
    *i *= 2;
}
```

The `pop()` method returns the last item of a vector wrapped in an `Option<T>` and removes the item from the vector:

```
let mut v = vec![1, 2, 3, 42];
if let Some(i) = v.pop() {
    println!("last: {}", i);
}
println!("rest: {} {} {}", v[0], v[1], v[2]);
```

Vectors can store different variants of an enum, for they belong to the same type:

```
enum DivisionResult {
    Integer(i32),
    FloatingPoint(f64),
    Undefined,
}

let results = vec![
    DivisionResult::Integer(3),          // 6/2
    DivisionResult::FloatingPoint(3.5), // 7/2
    DivisionResult::Undefined,           // 3/0
];

for r in results {
    match r {
        DivisionResult::Integer(i) => println!("{}", i),
        DivisionResult::FloatingPoint(f) => println!("{}", f),
        DivisionResult::Undefined => (),
    }
}
```

8.2 String

The `String` type is a collection of bytes which are interpreted as UTF-8 encoded text. Unlike the string slice type `str` (or the reference to it `&str`), `String` is not part of the core language, but implemented in the standard library.

A string can be created using the associated `new()` function, based on a string literal using the associated `from()` function, or using the `to_string()` method on an existing object implementing the `Display` trait:

```
let mut s1 = String::new();
let mut s2 = String::from("hello");
let mut s3 = "world".to_string();
```

Strings can be concatenated by using either the method `push_str()` (for adding a string slice) os `push()` (for a single character) on a mutable string:

```
let mut s = String::new();
s.push_str("hello");
s.push(' ');
s.push_str("world");
```

Two existing strings can be combined to a third string using the + operator.

```
let foo = String::from("foo");
let bar = String::from("bar");
let qux = foo + &bar;
println!("{}", qux);
println!("{}", foo); // error: foo was consumed!
println!("{}", bar); // ok: bar was not consumed
```

Internally, a method with the signature `add(self, s: &str) -> String` is called, which consumes the operand on the left, but doesn't own the operand on the right. No strings are copied; ownership of the first operand is taken and returned as the result of the concatenation.

Because chained concatenations are hard to read, the `format!` macro offers a more convenient interface, which also doesn't take ownership of any of the parameters:

```
let s1 = "hello".to_string();
let s2 = ", ".to_string();
let s3 = "world".to_string();
let message = format!("{}{}{}", s1, s2, s3);
```

Technically, the `String` type is a wraper over `Vec<u8>`. The `len()` method returns the number of *bytes* in the string, not *characters*:

```
let hello = String::from("hello");
let privet = String::from("привет");
println!("{}", hello.len());  // 5
println!("{}", privet.len()); // 12
```

The first string "hello" consists only of ASCII characters, which all can be encoded with a single byte in UTF-8. The cyrillic characters of the second string "привет", however, require two bytes in UTF-8.

Strings cannot be indexed in Rust, because a index might only be refering to some part of a encoded character, and the compiler prevents such error-prone operations. An indexing operation on the *characters* of a UTF-8 string would not be possible in constant time O(1), because a byte might only be *part of a character*, and the surrounding bytes needed to considered, too.

Slicing strings, however, is legal, but causes a panic if a slice starts or ends between two bytes belonging to the same character:

```
let privet = String::from("привет");
let pri = &privet[0..6];    // first three characters: при
let vet = &privet[6..12];   // last three characters: вет
println!("{}{}", pri, vet); // привет
let pri_ = &privet[0..7];   // panic: index 7 not a char boundary!
```

It's possible to iterate either over the bytes or the characters of a string:

```
let privet = String::from("привет");
for c in privet.chars() {
    print!("{}", c); // привет
}
println!();
for b in privet.bytes() {
    print!("{} ", b); // 208 191 209 128 208 184 208 178 208 181 209 130
}
```

8.3 Hash Map

A HashMap<K, V> stores a mapping of keys of type K to values of type V using a cryptographically safe hashing function for placement and lookup. Unlike a vector, the items cannot be retrieved by index, but by their unique key.

The type HashMap<K, V> is *not* part of the prelude and hence needs to be made available with use. A new hash map can be created using the associated new() method. Items can be added using the insert(k, v) method, which accepts a key and a value:

```
use std::collections::HashMap;

let mut points = HashMap::new();
points.insert(String::from("Myers"), 125);
points.insert(String::from("Roberts"), 99);
```

Keys and values can be of different types, like `String` and `i32` in the above example, but all keys and all values must be of the same type.

A common use case is to build up a hash map based on a list of keys and another list of values. This can be achieved by getting an iterator on each vector using the `iter()` method, zipping those iterators together using the `zip()` method and calling the `collect()` method on the result:

```
use std::collections::HashMap;

let names = vec!["Myers".to_string(), "Roberts".to_string()];
let score = vec![125, 99];
let mut points: HashMap<_, _> = names.iter().zip(score.iter()).collect();
```

The compiler needs a type annotation for `HashMap`, but can infer the key and value types.

Keys and values are owned by the hash map upon insertion:

```
let mut map = HashMap::new();
let key = String::from("Miller");
let val = 42;
map.insert(key, val);
println!("{}", key); // fail: key moved to map
```

Values of a `HashMap<K, V>` can be accessed using the `get()` method with a key, returning an `Option<&V>` containing a reference to the value found.

```
let mut results = HashMap::new();
results.insert("Peter".to_string(), 7.83);
results.insert("Michael".to_string(), 6.92);
results.insert("Paul".to_string(), 8.41);

if let Some(x) = results.get("Peter") {
    println!("Peter: {}", x);
}
```

An iteration over a hash map yields key-value tuples:

```
let mut results = HashMap::new();
results.insert("Peter".to_string(), 7.83);
results.insert("Michael".to_string(), 6.92);
results.insert("Paul".to_string(), 8.41);
```

```
for (key, value) in &results {
    println!("{}: {}", key, value);
}
```

There can only be stored one value per key, so calling `insert(key, value)` with a key already contained in the map overwrites that value.

```
let mut results = HashMap::new();
results.insert("Peter".to_string(), 7.83);
results.insert("Peter".to_string(), 6.92); // overwrite!

if let Some(x) = results.get("Peter") {
    println!("Peter: {}", x); // 6.92
}
```

Oftentimes, a new value should only be inserted into a hash map if a key is not contained in it yet. The `entry(key)` method takes a key and returns a `Entry`: a enum indicating an entry that might or might not exist. Its `or_insert(val)` method inserts a value into the hash map with the key given to the initial `entry(key)` call if an entry for that key was missing:

```
let mut results = HashMap::new();
results.insert("Peter".to_string(), 7.83);
results.insert("Michael".to_string(), 6.92);
results.insert("Paul".to_string(), 8.41);

results.entry("Phil".to_string()).or_insert(5.32); // new entry
results.entry("Paul".to_string()).or_insert(8.97); // existing entry
```

This interface is also useful for updating entries that already do exist. The `or_insert(0)` call in this example is only used to make sure that there is a starting value for the counter. The mutable reference returned by `to_insert(0)` is then used to increment the value, which must be dereferenced:

```
let text = "He had had had a answer";
let mut wordcount = HashMap::new();

for word in text.split_whitespace() {
    let count = wordcount.entry(word).or_insert(0);
    *count += 1;
}
```

9 Error Handling

Rust has no exceptions. Instead, it provides two ways of dealing with different categories of errors: the `panic!` macro for unrecoverable errors, and the type `Result<T, E>` for recoverable errors.

9.1 Unrecoverable Errors: `panic!`

If an error condition occurs that cannot be recovered from, calling the `panic!` macro (with a sensible error message, indicating the underlying issue) will print the error message, undwind and clean up the stack and quit the program:

```
fn main() {
    panic!("Something terrible happened!");
}
```

The output of the program will indicate the error:

```
    Finished dev [unoptimized + debuginfo] target(s) in 0.00s
     Running `target/debug/tmp`
thread 'main' panicked at 'Something terrible happened!', src/main.rs:2:5
note: Run with `RUST_BACKTRACE=1` environment variable to display a backtrace.
```

The cause of the error is not always that easy to track down. In this case, executing the program with backtrace enabled (as suggested in the output above), the whole stack trace will be printed out upon panicking:

```
$ RUST_BACKTRACE=1 cargo run
    Finished dev [unoptimized + debuginfo] target(s) in 0.01s
     Running `target/debug/tmp`
thread 'main' panicked at 'Something terrible happened!', src/main.rs:2:5
stack backtrace:
   0: std::sys::unix::backtrace::tracing::imp::unwind_backtrace
             at src/libstd/sys/unix/backtrace/tracing/gcc_s.rs:39
   1: std::sys_common::backtrace::_print
             at src/libstd/sys_common/backtrace.rs:71
   2: std::panicking::default_hook::{{closure}}
             at src/libstd/sys_common/backtrace.rs:59
             at src/libstd/panicking.rs:197
   3: std::panicking::default_hook
             at src/libstd/panicking.rs:211
   4: std::panicking::rust_panic_with_hook
             at src/libstd/panicking.rs:474
   5: std::panicking::begin_panic
          at /rustc/a53f9df32fbb0b5f4382caaad8f1a46f36ea887c/src/libstd/panicking.rs:408
   6: tmp::main
             at src/main.rs:2
   7: std::rt::lang_start::{{closure}}
             at /rustc/a53f9df32fbb0b5f4382caaad8f1a46f36ea887c/src/libstd/rt.rs:64
   8: std::panicking::try::do_call
             at src/libstd/rt.rs:49
             at src/libstd/panicking.rs:293
   9: __rust_maybe_catch_panic
             at src/libpanic_unwind/lib.rs:85
  10: std::rt::lang_start_internal
             at src/libstd/panicking.rs:272
             at src/libstd/panic.rs:394
             at src/libstd/rt.rs:48
  11: std::rt::lang_start
             at /rustc/a53f9df32fbb0b5f4382caaad8f1a46f36ea887c/src/libstd/rt.rs:64
  12: main
  13: __libc_start_main
  14: _start
```

In order to find the erroneous code of your own, start reading the backtrace at the top and follow along until you see a source file you've written on your own (above: output line 6: `src/main.rs:2`).

Instead of unwinding and cleaning up the stack upon panicking, the program can also abort immediately, leaving the cleanup task up to the operating system. This can be configured in the respective `[profile]` section (debug, release) in the project's `Cargo.toml` file:

```
[profile.release]
panic = 'abort'
```

9.2 Recoverable Errors: `Result<T, E>`

Most error conditions do not require the program to stop, but only to take a different path in the program logic. The result of a potentially failing operation can be expressed using the `Result<T, E>` enum, which is defined as such:

```
enum Result<T, E> {
    Ok(T),
    Err(E),
}
```

An operation that couldl be executed as intended has the `Ok` variant set (with any appropriate type), whereas a failing operation has the `Err` variant set (with some error type).

For example, the method `File::open(filename)` returns a `Result<T, E>` with `T=std::fs::File` set in case of success, and `E=std::io::Error` set in case of an error.

The result can be handled using a `match` expression, panicking in the error case:

```rust
use std::fs::File;

fn main() {
    let f = File::open("hello.txt");
    let f = match f {
        Ok(file) => file,
        Err(error) => panic!("Error opening file 'hello.txt': {:?}", error),
    };
}
```

Since there are different possible causes for a file not being able to be opened, it is more appropriate to differentiate the error occured accordingly. The `kind()` method of the error can be used for this purpose, and the *match guard* syntax helps to prevent deeply indented error handling code blocks:

```rust
use std::fs::File;
use std::io::ErrorKind;

fn main() {
    let f = File::open("hello.txt");
    let f = match f {
        Ok(file) => file,
        Err(ref error) if error.kind() == ErrorKind::NotFound => match File::create("hello.txt") {
            Ok(fc) => fc,
            Err(e) => panic!("Error creating file 'hello.txt': {:?}", e),
        },
        Err(error) => panic!("Error opening file 'hello.txt': {:?}", error),
    };
}
```

The `ref` keyword ensures that the error value is not moved inside the `match` arm, but referred to instead.

9.2.1 `unwrap` and `expect`

The `unwrap` method of the `Result<T, E>` type is a shortcut for error handling. If the `Ok` variant is set, `unwrap` returns the value set to the `Ok` variant; if the `Err` variant is set, the program will cause a panic with the default error message:

```
use std::fs::File;

fn main() {
    let f = File::open("hello.txt").unwrap();
}
```

The `expect` method works just like `unwrap`, but accepts a string parameter indicating a custom error message:

```
use std::fs::File;

fn main() {
    let f = File::open("hello.txt").expect("error opening 'hello.txt'");
}
```

9.2.2 Error Propagation

It is not always possible or desirable to deal with every error condition where it originally occurs. Oftentimes, the callee does not know the context, and the caller should decide on how to deal with the error. This technique is called *error propagation*:

```
use std::fs::File;
use std::io;
use std::io::Read;

fn read_str_from_file() -> Result<String, io::Error> {
    let f = File::open("hello.txt");
    let mut f = match f {
        Ok(file) => file,
        Err(e) => return Err(e), // propagate error
    };
    let mut s = String::new();
    match f.read_to_string(&mut s) {
        Ok(_) => Ok(s),
        Err(e) => Err(e), // propagate error
    }
}

fn main() {
    match read_str_from_file() {
        Ok(_) => println!("ok"),
        Err(_) => panic!("not good"),
    }
}
```

This pattern is so common that Rust offers the shortcut operator `?` for propagating error. The same code becomes much shorter using that syntactic sugar:

```rust
use std::fs::File;
use std::io;
use std::io::Read;

fn read_str_from_file() -> Result<String, io::Error> {
    let mut f = File::open("hello.txt")?; // ? instead of match
    let mut s = String::new();
    f.read_to_string(&mut s)?; // ? instead of match
    Ok(s)
}

fn main() {
    match read_str_from_file() {
        Ok(_) => println!("ok"),
        Err(_) => panic!("not good"),
    }
}
```

The `?` operator works as follows: If the expression in front of it evaluates to the `Ok` variant of `Result`, the `Ok` variant is returned. If it evaluates to the `Err` variant instead, the `Err` variant is returned.

One difference between the two implementations above is that `?` causes a call to the `from` function of the `From` trait to convert the error type to the defined return type (`io::Error` in this case).

The same function can be made even shorter by chaining the method calls after the `?` operator:

```rust
fn read_str_from_file() -> Result<String, io::Error> {
    let mut s = String::new();
    File::open("hello.txt")?.read_to_string(&mut s)?;
    Ok(s)
}
```

The `?` operator can only be used in functions that return a `Result<T, E>`.

9.3 Validation Types

Using validation functions throughout the code to prevent errors can be tedious and lead to repeated code. Restrictions, say, on the value range of variables, can be better expressed by types.

The builtin `u8` type, for example, is the perfect choice if discrete numbers from 0 to 255 are the only acceptable values. However, there's not a builtin type for every restriction needed. (Like a program for a lottery game, which only accepts numbers between 1 and 45.) In this case, it's better to implement a new custom type.

Implementing a custom validation type requires three things:

1. A private value variable with an appropriate range, enclosing the possible value range.
2. A public `new` method containing the validation code, which accepts a value parameter to be checked.
3. A public `value` method, returning the underlying value.

For the lottery game, a custom validation type can be implemented as follows:

```rust
pub struct Guess {
    value: u8,
}

impl Guess {
    pub fn new(value: u8) -> Guess {
        if value < 1 || value > 45 {
            panic!("acceptable range: 1..45");
        }
        Guess { value }
    }

    pub fn value(&self) -> u8 {
        self.value
    }
}

fn main() {
    Guess::new(10); // ok
    Guess::new(0); // not ok
    Guess::new(50); // not ok
}
```

9.4 Using `panic!` or `Result<T, E>`?

Since there's no way to recover from a panic, working with `Result<T, E>` is isually the more appropriate choice. The use of `panic!` (and the methods `unwrap` and `expect`, which also cause a panic) should be restricted to the following situations:

1. When a test case that needs to fail as a whole.
2. If an error is logically impossible, but the compiler cannot figure that out.

3. If the code could end up in a bad state that cannot be recovered from, leading to further undefined states. This usually happens when the caller is using an API the wrong way (contract violation). An API causing a panic must always mention that in its documentation.
4. For example code and prototypes.

10 Traits, Generics and Lifetimes

Traits and generics help programmers write more flexible code that can be re-used instead of being copied and (slightly) modified all the time. Lifetimes ensure that no invalid references can occur, which is one of the pillars of Rust's runtime safety.

10.1 Traits

Traits are Rust's way of declaring common functionality (a set of method signatures) that can be implemented by different types.

A trait is defined using the `trait` keyword, following a list of method signatures:

```rust
pub trait Summary {
    fn summarize(&self) -> String;
}
```

The trait is public, so that it can be implemented in other crates as well.

Given the structs `NewsArticle` and `Tweet`:

```rust
pub struct NewsArticle {
    pub headline: String,
    pub location: String,
    pub author: String,
    pub content: String,
}

pub struct Tweet {
    pub username: String,
    pub content: String,
    pub reply: bool,
    pub retweet: bool,
}
```

The `Summary` trait can be implemented for them using the `for` keyword:

```
impl Summary for NewsArticle {
    fn summarize(&self) -> String {
        format!("{}, by {} ({})", self.headline, self.author, self.location)
    }
}

impl Summary for Tweet {
    fn summarize(&self) -> String {
        format!("{}: {}", self.username, self.content)
    }
}
```

The `summarize` method can then be used like this:

```
fn main() {
    let n = NewsArticle {
        headline: "Sack of Rice Fell Down".to_string(),
        location: "China".to_string(),
        author: "Russel F. Important".to_string(),
        content: "Well, the headline says it all...".to_string(),
    };
    let t = Tweet {
        username: "@russelfimportant".to_string(),
        content: "Sack of Rice fell down in China.".to_string(),
        reply: false,
        retweet: false,
    };
    println!("NewsArticle: {}", n.summarize());
    println!("Twitter: {}", t.summarize());
}
```

Output:

```
NewsArticle: Sack of Rice Fell Down, by Russel F. Important (China)
Twitter: @russelfimportant: Sack of Rice fell down in China.
```

A trait can only be implemented if either the trait or the type is local to the crate. External traits cannot be implemented on external types.

10.1.1 Default Implementations

A trait can define a default behaviour for a method, i.e. providing a method implementation that can but does not has to be overwritten. If the default implementation is not to be overwritten, the trait can be implemented with an empty `impl` block:

```rust
pub trait Summary {
    fn summarize(&self) -> String {
        String::from("(Read more...)")
    }
}

pub struct BlogPost {
    pub title: String,
    pub url: String,
    pub author: String,
}

impl Summary for BlogPost {}
```

Default implementations can call other methods of the same trait, even those without a default implementation. It is not possible to call the default implementation from an overriding implementation.

10.2 Generics

Instead of writing the same data structure or function over and over for different types (say, `i32` and `f32` for a mathematics library that supports both integer and floating point arithmetic), types can be parametrized using *generics*.

10.2.1 Generic Structs

The two structs `PointDiscrete` and `PointContinuous` define the same fields (`x` and `y`), but use different type for those: integers and floats.

```rust
struct PointDiscrete {
    x: i32,
    y: i32,
}

struct PointContinuous {
    x: f32,
    y: f32,
}
```

Those two types can almost be used alike, but for the type system, there something fundamentally different.

```
fn main() {
    let a = PointDiscrete { x: 12, y: 7 };
    let b = PointContinuous { x: 3.75, y: 2.12 };
    println!("a=({}, {}), b=({}, {})", a.x, a.y, b.x, b.y);
}
```

Introducing a generic type parameter `T` not only helps merging the two struct definitions to one, but also allows using other types, such as `f64` or `u16`:

```
struct Point<T> {
    x: T,
    y: T,
}

fn main() {
    let a = Point { x: 12, y: 7 };
    let b = Point { x: 3.75, y: 2.12 };
    println!("a=({}, {}), b=({}, {})", a.x, a.y, b.x, b.y);
}
```

The actual types of point `a` and `b` are inferred. Since there's only a single type parameter defined, and `T` is the same for the field `x` and `y`, the following code won't compile:

```
let c = Point { x: 10, y: 3.14 };
println!("c=({}, {})", c.x, c.y);
```

The compiler infers some integer type for `x`, so `y` must also be an integer:

error[E0308]: mismatched types –> src/main.rs:11:31 | 11 | let c = Point { x: 10, y: 3.14 }; | ^^^^ expected integer, found floating-point number | = note: expected type `{integer}` found type `{float}`

If two type parameters, `T` und `U`, are used for the struct definitions, the above code compiles, no matter if `T` and `U` are the same types or different:

```
struct Point<T, U> {
    x: T,
    y: U,
}

fn main() {
    let a = Point { x: 12, y: 7 };
    let b = Point { x: 3.75, y: 2.12 };
    println!("a=({}, {}), b=({}, {})", a.x, a.y, b.x, b.y);

    let c = Point { x: 10, y: 3.14 };
    println!("c=({}, {})", c.x, c.y);
}
```

However, using different types for `x` and `y` could make it harder to use common operations on those fields: The less the compiler knows about a type, the fewer operations can be performed on fields of that type. Therefore generic type parameters should only be used if it clearly serves the use case at hand, and not for flexibility for it's own sake.

10.2.2 Generic Enums

Rust's two most common enums, `Option<T>` and `Result<T, E>` use generic type parameters:

```rust
enum Option<T> {
    Some(T),
    None,
}
```

`Option` only has a single type parameter, `T`. The `None` case does not need a type, because it is used to signify the absence of a value, and hence the absence of a type.

```rust
enum Result<T, E> {
    Ok(T),
    Err(E),
}
```

`Result`, however, is parametrized with two types, `T` and `E`; the `E` parameter is used for some error type, whereas the `T` parameter specifies the type for the result of a successful operation, which usually is not an error message, even though this enum definition does not forbid using it that way.

10.2.3 Generic Functions

The two functions `largest_i` and `largest_f` take a list of integers (`i32`) or floats (`f32`), respectively, find the biggest item in the list and return it:

```rust
fn largest_i(list: &[i32]) -> i32 {
    let mut largest = list[0];

    for &item in list.iter() {
        if item > largest {
            largest = item
        }
    }
    largest
}

fn largest_f(list: &[f32]) -> f32 {
    let mut largest = list[0];

    for &item in list.iter() {
        if item > largest {
            largest = item
        }
    }

    largest
}

fn main() {
    let integers = vec![53, 12, 76, 19, 44, 98, 72];
    let floats = vec![1.45, 93.9, 64.3, 91.3, 45.1];

    println!("largest of {:?}: {}", integers, largest_i(&integers));
    println!("largest of {:?}: {}", floats, largest_f(&floats));
}
```

Even though the method only differs in the signature and body (the implementation) is exactly the same twice, the whole code has been duplicated.

Using generics, type information can be parametrized to avoid such duplication. This implementation replaces the specific types `i32` and `f32` with a generic type parameter `T` (a common convention for "type"):

```
fn largest<T>(list: &[T]) -> T { // T instead of i32/f32
    // implementation unchanged
}
```

Unfortunately, this method does not compile:

```
error[E0369]: binary operation `>` cannot be applied to type `T`
  --> src/main.rs:29:17
   |
29 |         if item > largest {
   |            ---- ^ ------- T
   |            |
   |            T
   |
   = note: `T` might need a bound for `std::cmp::PartialOrd`
```

Comparisons with the > operator do not work for *all* types, but only for types that implement the trait `std::cmp::PartialOrd` (as the compiler suggests).

10.2.4 Trait Bounds

The types that can be used for a generic type parameter can be constrained using a *trait bound*. The type parameter (`<T>`) is supplied with a trait name (`<T: Trait>`), so that the function is only appliable to types satisfying the given trait. It's also possible to constrain a type using multiple traits by separating those traits with a `+`, such as `<T: Trait1 + Trait2>`.

The `largest` function from above can be made to work by constraining the type parameter `T` to the traits `PartialOrd` (for the > comparison) and `Copy` (so that the items from the list are copied and not moved out of the list):

```rust
fn largest<T: PartialOrd + Copy>(list: &[T]) -> T {
    let mut largest = list[0];

    for &item in list.iter() {
        if item > largest {
            largest = item
        }
    }

    largest
}
```

An alternate syntax using the `where` clause helps keeping the function signature short:

```rust
fn largest<T>(list: &[T]) -> T
where
    T: PartialOrd + Copy,
{
    let mut largest = list[0];

    for &item in list.iter() {
        if item > largest {
            largest = item
        }
    }

    largest
}
```

The difference is more striking if multiple type parameters are involved:

```rust
fn set<K: Display + Copy, V: PartialOrd + Copy>(key: K, value: V) {
    // ...
}
```

Compared to:

```rust
fn set<K, V>(key: K, value: V)
where
    K: Display + Copy,
    V: PartialOrd + Copy,
{
    // ...
}
```

10.2.5 Generic Methods

To implement a method for all possible types on a struct with a type parameter, the type parameter must also be declared for the `impl` block:

```rust
struct Point<T> {
    x: T,
    y: T,
}

impl<T> Point<T> {
    fn x(&self) -> &T {
        &self.x
    }
    fn y(&self) -> &T {
        &self.y
    }
}
```

It is also possible to implement certain methods only for specific types, say, floating point numbers, which support additional mathematical operations (e.g. square roots) compared to integers:

```rust
impl Point<f32> {
    fn distance_from_origin(&self) -> f32 {
        (self.x.powi(2) + self.y.powi(2)).sqrt()
    }
    fn distance_from(&self, other: &Point<f32>) -> f32 {
        ((self.x - other.x).powi(2) + (self.y - other.y).powi(2)).sqrt()
    }
}

fn main() {
    let p = Point { x: 3.75, y: 2.12 };
    let q = Point { x: 1.5, y: 8.95 };

    println!("distance p to (0, 0): {}", p.distance_from_origin());
    println!("distance q to (0, 0): {}", q.distance_from_origin());
    println!("distance p to q: {}", p.distance_from(&q));
    println!("distance q to p: {}", q.distance_from(&p));
}
```

The methods can be called on any `Point` using `f32` values. However, if a point uses integer values, the methods above are not available:

```
fn main() {
    let a = Point { x: 1, y: 2 };
    println!("distance a to (0, 0): {}", a.distance_from_origin())
}
```

Output:

```
error[E0599]: no method named `distance_from_origin` found for type `Point<{integer}>` in the current scope
  --> src/main.rs:17:44
   |
1  | struct Point<T> {
   | --------------- method `distance_from_origin` not found for this
...
17 |     println!("distance a to (0, 0): {}", a.distance_from_origin())
   |                                            ^^^^^^^^^^^^^^^^^^^^
```

The type parameters of a method parameter are not restricted to the type parameters of the `impl` block. Starting from this struct definition with different types for `x` and `y`:

```
struct Point<T, U> {
    x: T,
    y: U,
}
```

A method can be implemented that works on a `Point` with certain type parameters, while accepting `Point` parameters that use (possibly) different type parameters:

```
impl<T, U> Point<T, U> {
    fn mixup<V, W>(self, other: Point<V, W>) -> Point<T, W> {
        Point {
            x: self.x,  // T
            y: other.y, // W
        }
    }
}

fn main() {
    let a = Point { x: 1, y: 2.5 };
    let b = Point { x: 3.99, y: 8 };
    let p = a.mixup(b);
    println!("p=({},{})", p.x, p.y); // Output: p=(1,8)
}
```

The type parameters `T` and `U` are the types of the callee (`self`), whereas `V` and `W` are the types of the method parameter (`other`). The method `mixup` produces a new point with mixed-up type parameters. For this purpose, type parameters from both the `impl` block and the method signature can be used.

10.2.6 Conditionally Implement Methods

Methods can be implemented conditionally for types that satisfy specific traits. The `cmp_display` method can only be invoked on a `Pair<T>` whose `T` implements both the `Display` and the `PartialOrd` trait:

```
use std::fmt::Display;

struct Pair<T> {
    x: T,
    y: T,
}

impl<T: Display + PartialOrd> Pair<T> {
    fn cmp_display(&self) {
        if self.x >= self.y {
            // PartialOrd
            println!("The largest member is x={}", self.x); // Display
        } else {
            println!("The largest member is y={}", self.y); // Display
        }
    }
}

fn main() {
    let p = Pair { x: 3, y: 7 };
    p.cmp_display();
}
```

Since integer types satisfy both traits, the code above compiles, and the `cmp_display` method can be called on the `Pair p`.

It is also possible to implement a trait for any type that implements another trait. Let's say that the method `identify` should implemented for all the types that satisfy the `Display` trait:

```
use std::fmt::Display;

trait Subject {
    fn identify(&self);
}

impl<T: Display> Subject for T {
    fn identify(&self) {
        println!("I am {}.", self);
    }
}

fn main() {
    let pi = 3.14;
    pi.identify(); // I am 3.14.
}
```

The `identify()` method is now available for all other types satisfying `Display`, without providing any futher implementations. Such implementations are called *blanket implementations*. They are documented in the "Implementors" section to each trait.

10.2.7 Compilation and Performance

The Rust compiler turns generic code into code with specific types upon compilation (_monomorphization). If a struct `Point<T>` is used with the specific types `i32` and `f32`, the compiler will fill in the types and produce two implementations: `Point_i32` and `Point_f32`. For any usage of some `Point<T>`, the type parameter will be inferred, and the code will be compiled using the specific types.

This comes at some cost: Not only is the compilation process more complicated and therefore slower, but also is the resulting code gets bigger. However, handling generics at compile time ensures that there are no runtime costs; and because a programm is usually compiled once and run many times, the additional compilation time can be seen as a investment rather than a cost.

10.3 Lifetimes

Every reference in Rust has a *lifetime*. This is a scope for which that reference is valid. As soon as the end of that scope is reached, the lifetime expires, and the reference becomes invalid.

10.3.1 Dangling References

A reference to a value that went out of scope is called a *dangling reference*. Such a reference is invalid and must not be used any longer, which is enforced by the Rust compiler. The following program won't compile:

```
fn main() {
    let r;
    {
        let x = 5;
        r = &x;
    } // here, x goes out of scope
    println!("r: {}", r);
}
```

Error message:

```
   Compiling tmp v0.1.0 (/home/paedu/learning-rust/code/tmp)
error[E0597]: `x` does not live long enough
 --> src/main.rs:5:9
  |
5 |         r = &x;
  |         ^^^^^^ borrowed value does not live long enough
6 |     }
  |     - `x` dropped here while still borrowed
7 |     println!("r: {}", r);
  |                       - borrow later used here
```

The *borrow checker* compares the scopes involved and notices that the subject of the reference (x) does not live as long as the reference (r).

A simplified version of the program without the inner scope will compile:

```
fn main() {
    let x = 5;
    let r = &x;
    println!("r: {}", r);
}
```

The reference r and the subject x have the same scope and, hence, the same lifetime.

10.3.2 Lifetimes in Functions

Consider this program that figures out which of two strings is the longer one:

```
fn main() {
    let a = String::from("foobar");
    let b = String::from("qux");

    let result = longest(a.as_str(), b.as_str());
    println!("The longest string is '{}'", result);
}
```

It uses the `longest` function to do the actual work:

```
fn longest(x: &str, y: &str) -> &str {
    if x.len() > y.len() {
        x
    } else {
        y
    }
}
```

This code doesn't compile:

```
error[E0106]: missing lifetime specifier
 --> src/main.rs:9:33
  |
9 | fn longest(x: &str, y: &str) -> &str {
  |                                 ^ expected lifetime parameter
  |
  = help: this function's return type contains a borrowed value, but the
          signature does not say whether it is borrowed from `x` or `y`
```

The compiler cannot know if the reference being returned from the function will be borrowed from `x` or from `y`. This is an issue, because the borrow checker is unable to figure out the valid scope for the returned reference, since `x` and `y` could have different lifetimes,

Lifetime annotations describe the relationships of the lifetimes of multiple references. They are just a hint to the borrow checker, and do not change the lifetime or scope of any reference.

The syntax of lifetime annotations looks as follows:

```
&i32        // a reference without an annotated lifetime
&'a i32     // a reference with the explicit lifetime a
&'b mut i32 // a mutable reference with the explicit lifetime b
```

The lifetime parameter also needs to be declared in angle brackets after the function name. The following version of the longest function annotates both function parameters and the returned reference with the same lifetime:

```rust
fn longest<'a>(x: &'a str, y: &'a str) -> &'a str {
    if x.len() > y.len() {
        x
    } else {
        y
    }
}
```

The method signature can be read as: “The function longest has a lifetime parameter a; the parameters x and y are both references with the same lifetime a, and the reference returned has also the same lifetime a.”

This code compiles, and now the borrow checker knows how to enforce the lifetimes of the references passed into this function.

```rust
fn main() {
    let result: &str;
    let a = String::from("foobar");
    {
        let b = String::from("qux");
        result = longest(a.as_str(), b.as_str());
    }
    println!("The longest string is '{}'", result);
}
```

Error message:

```
error[E0597]: `b` does not live long enough
 --> src/main.rs:6:38
  |
6 |         result = longest(a.as_str(), b.as_str());
  |                                      ^ borrowed value does not live long enough
7 |     }
  |     - `b` dropped here while still borrowed
8 |     println!("The longest string is '{}'", result);
  |                                           ------ borrow later used here
```

The borrow checker sees that a, b and result are supposed to have the same lifetime, but also notices that b has a shorter scope than the other references. Not considering the actual values behind a and b, the compiler does not know that result will be referring to a in this specific example, so potentially a dangling pointer could result from this code. The compilation therefore fails.

Consider a slightly modified version of the above client code:

```
fn main() {
    let result: &str;
    let a = String::from("foobar");
    {
        let b = String::from("qux");
        result = longest(a.as_str(), b.as_str());
        println!("The longest string is '{}'", result);
    }
}
```

This code compiles, even though the three variables involved have a different scope. This is because the generic lifetime `'a` will be the *smaller* lifetime of the two parameters `x` and `y`, and all references are used within that smaller scope.

10.3.3 Lifetime Annotations in Structs

Structs can hold references:

```
struct Excerpt {
    part: &str,
}

fn main() {
    let text = String::from("This is important. Or maybe not...");
    let first_sentence = text.split('.').next().expect("no . found");
    let excerpt = Excerpt {
        part: first_sentence,
    };
    println!("{}", excerpt.part);
}
```

However, this code does not compile, because the reference has no lifetime specified:

```
error[E0106]: missing lifetime specifier
 --> src/main.rs:2:11
  |
2 |     part: &str,
  |           ^ expected lifetime parameter
```

This is an issue, because the struct as a whole could outlive the reference it's holding, leading to a dangling pointer. A lifetime stating that both the struct and its field have the same lifetime fixes the problem:

```
struct Excerpt<'a> {
    part: &'a str,
}
```

10.3.4 Lifetime Elision

Not every function returning a reference to one of its parameters needs lifetime annotations. Consider this function, which returns the first word of a given string as a reference:

```
fn first_word(s: &str) -> &str {
    let bytes = s.as_bytes();
    for (i, &item) in bytes.iter().enumerate() {
        if item == b' ' {
            return &s[0..i]; // return part up to first space
        }
    }
    &s[..] // no spaces: s is a single word
}

fn main() {
    let text = String::from("Rust kicks derrieres").to_string();
    let word = first_word(&text);
    println!("{}", word); // Rust
}
```

This code compiles, even though there are no lifetimes specified. The compiler has a set of rules called *lifetime elision rules*. If those apply to the code in question, i.e. to the parameter's *input lifetimes* and to the resulting references *output lifetimes*, the lifetimes can be inferred from the context. Those rules are:

1. Every input parameter gets its own lifetime parameter.
2. If there is one input lifetime parameter, the output lifetime parameter is assigned this input lifetime parameter.
3. If there are multiple input lifetime parameters, but one of them is `&self` or `&mut self` (the function is a method), the lifetime of `self` is assigned to all output lifetime parameters.

10.3.5 Static Lifetime

There is one special lifetime parameter: the `'static` lifetime. This donates that a value is in scope for the entire lifetime of the program. String literals, which are stored in the binary upon compilation, have the `'static` lifetime.

This code is valid, because the string literal lives long enough (for the whole runtime of the program that is).

```
let s: &'static str = "This is a question.";
println!("{}", s);
```

However, this code does not compile:

```
let j = 123;
let i: &'static i32 = &j;
println!("{}", i);
```

j only lives as long as the enclosing scope, and i, the reference to it, is supposed to live for the duration of the whole program:

```
error[E0597]: `j` does not live long enough
 --> src/main.rs:6:27
  |
6 |     let i: &'static i32 = &j;
  |            ------------   ^^ borrowed value does not live long enough
  |            |
  |            type annotation requires that `j` is borrowed for `'static`
7 |     println!("{}", i);
8 | }
  | - `j` dropped here while still borrowed
```

10.3.6 Generics, Traits, and Lifetimes Combined

The function longest_print uses all the concepts covered in this chapter: generics, traits and lifetimes:

```
use std::fmt::Display;

fn longest_print<'a, T>(x: &'a str, y: &'a str, caller: T) -> &'a str
where
    T: Display,
{
    println!("longest_print called from {}", caller);
    if x.len() > y.len() {
        x
    } else {
        y
    }
}

fn main() {
    let a = &String::from("hello");
    let b = &String::from("Rust");
    let r = longest_print(&a, &b, "main");
    println!("longest: '{}'", r);
}
```

- The generic type parameter `T` for the function parameter `caller`.
- The trait `Display` to constrain the parameter `T`, which must be printable.
- The lifetime `'a` to ensure that the resulting reference does not outlive the parameter references.

11 Writing Automated Tests

Rust has built-in support for automated tests. A test module and a test function is automatically generated for every new library crate:

```
$ cargo new maths --lib
```

The generated code:

```
#[cfg(test)]
mod tests {
    #[test]
    fn it_works() {
        assert_eq!(2 + 2, 4);
    }
}
```

The test module is annotated with `[#cfg(test)]`, and the test function with `[#test]`. The function body tests if `2 + 2` equals `4`, which always holds true, so the test case passes when run using `cargo test` (excerpt of the output):

```
$ cargo test

running 1 test
test tests::it_works ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out
```

A test fails if a test function (or one of the function it calls) panics:

```
#[cfg(test)]
mod tests {
    #[test]
    fn it_panics() {
        panic!("nothing works yet");
    }
}
```

Output of `cargo test` (excerpt):

```
running 1 test
test tests::it_panics ... FAILED

failures:

---- tests::it_panics stdout ----
thread 'tests::it_panics' panicked at 'nothing works yet', src/lib.rs:5:9
note: Run with `RUST_BACKTRACE=1` environment variable to display a backtrace.


failures:
    tests::it_panics

test result: FAILED. 0 passed; 1 failed; 0 ignored; 0 measured; 0 filtered out
```

11.1 Assertions

The `assert!` macro ensures that an expression evaluates to `true`. If the expression evaluates to `false`, the `panic!` macro is called and the test case fails.

This test case checks the return value of the `add` function. The test module needs to import the function from its super module:

```
pub fn add(a: i32, b: i32) -> i32 {
    a + b
}

#[cfg(test)]
mod tests {
    use super::add;

    #[test]
    fn test_add() {
        // arrange
        let a = 7;
        let b = 3;
        let expected_sum = 10;

        // act
        let actual_sum = add(a, b);

        // assert
        assert!(actual_sum == expected_sum);
    }
}
```

The test function is built up using the “AAA” pattern, which stands for:

1. Arrange: set up the test data
2. Act: run the code to be tested
3. Assert: check the outcomes against the expectations

The macros `assert_eq!` is especially helpful for comparisons. It takes two parameters, called `left` and `right`, checks them for equality.

This broken implementation of the `subtract` function shows the difference between `assert!` and `assert_eq!`:

```
pub fn sub(a: i32, b: i32) -> i32 {
    b - a // broken: must be a - b
}

#[cfg(test)]
mod tests {
    use super::sub;

    #[test]
    fn test_subtract1() {
        assert!(7 == sub(10, 3))
    }

    #[test]
    fn test_subtract2() {
        assert_eq!(7, sub(10, 3))
    }
}
```

Output of `cargo test`:

```
running 2 tests
test tests::test_subtract1 ... FAILED
test tests::test_subtract2 ... FAILED

failures:

---- tests::test_subtract1 stdout ----
thread 'tests::test_subtract1' panicked at 'assertion failed: 7 == sub(10, 3)', src/lib.rs:30:9
note: Run with `RUST_BACKTRACE=1` environment variable to display a backtrace.

---- tests::test_subtract2 stdout ----
thread 'tests::test_subtract2' panicked at 'assertion failed: `(left == right)`
  left: `7`,
 right: `-7`', src/lib.rs:35:9


failures:
    tests::test_subtract1
    tests::test_subtract2

test result: FAILED. 0 passed; 2 failed; 0 ignored; 0 measured; 0 filtered out
```

In `subtract1`, using the `assert!` macro, the unevaluated expression is shown. In `subtract2`, using the `assert_eq!` macro, the evaluated `left` and `right` expressions are shown, making error detection arguably easier.

The `assert_ne!` macro ensures that the `left` and `right` expression are *not* equals. Sometimes, it is not possible to say what the outcome of an operation should be, but it's well possible to say what the outcome must *not* be.

11.2 Custom Failure Messages

The macros `assert!`, `assert_eq!` and `assert_ne!` take an optional third parameter: a custom error message that is printed in case the assertion fails:

```rust
pub fn add(a: i32, b: i32) -> i32 {
    a + b // correct
}

pub fn sub(a: i32, b: i32) -> i32 {
    b - a // broken
}
```

```rust
#[cfg(test)]
mod tests {
    use super::add;
    use super::sub;

    #[test]
    fn test_add() {
        assert_eq!(10, add(7, 3), "adding 3 to 7 failed")
    }

    #[test]
    fn test_subtract() {
        assert_eq!(7, sub(10, 3), "subtracting 3 from 10 failed")
    }
}
```

Output (excerpt):

```
running 2 tests
test tests::test_add ... ok
test tests::test_subtract ... FAILED

failures:

---- tests::test_subtract stdout ----
thread 'tests::test_subtract' panicked at 'assertion failed: `(left == right)`
  left: `7`,
 right: `-7`: subtracting 3 from 10 failed', src/lib.rs:21:9
note: Run with `RUST_BACKTRACE=1` environment variable to display a backtrace.


failures:
    tests::test_subtract

test result: FAILED. 1 passed; 1 failed; 0 ignored; 0 measured; 0 filtered out

error: test failed, to rerun pass '--lib'
```

The custom error message of the failing `test_subtract` case is shown in the output.

11.3 Testing for Panics

As stated in the chapter about error handling, a panic is a valid response to a public interface that is used the wrong way. The `#[should_panic]` annotation helps testing such cases: The test case passes if a panic is caused upon execution, and fails otherwise.

This `div` function panics if the given divisor is equal to zero, and the panic is expected in the test function:

```rust
pub fn div(a: i32, b: i32) -> f32 {
    if b == 0 {
        panic!("division by zero");
    }
    a as f32 / b as f32
}

#[cfg(test)]
mod tests {
    use super::div;

    #[test]
    #[should_panic]
    fn test_divide_by_zero() {
        div(3, 0);
    }
}
```

The test passes, but it would also pass if the panic happened for a different reason. For such cases, the optional `expected` parameter of the `should_panic` annotation helps to ensure that the underlying code panics for the right reason by checking if the `expected` parameter is a substring of the actual panic message:

```rust
#[test]
#[should_panic(expected="division by zero")]
fn test_divide_by_zero() {
    div(3, 0);
}
```

This approach is somewhat shaky, because panic messages can change as time goes. The `expected` parameter therefore must be chosen to be neither too specific (danger of breaking with changes) nor too general (danger of accepting wrong panic messages).

11.4 Test Execution

11.4.1 Arguments

The command `cargo test` compiles the project in test mode and runs the resulting test binary. Command line parameters can be indicated both for cargo and the test binary. For the latter case, parameters following the -- separator are sent to the test binary:

```
cargo test --foo # send parameter --foo to cargo
cargo test -- --bar #  send parameter --bar to the test binary
cargo test --foo -- --bar # send --foo to cargo, --bar to the test binary
```

11.4.2 Parallel Execution

By default, multiple tests run in multiple threads: one test per thread, that is. The order of test execution is not deterministic, and therefore tests should not depend on one another. However, the number of threads can be defined using the `--test-threads` parameter, which can be set to `1` for sequential execution.

```
cargo test -- --test-threads=1
```

The standard output of passing tests won't be shown in the test result. Consider this two test cases:

```
#[cfg(test)]
mod tests {
    #[test]
    fn passing_test() {
        println!("is 2 + 2 = 4?");
        assert_eq!(2 + 2, 4);
    }

    #[test]
    fn failing_test() {
        println!("is 2 + 2 = 5?");
        assert_eq!(2 + 2, 5);
    }
}
```

11.4.3 Output

When run with `cargo test`, only the `println!` output of the failing test is shown in the `stdout` section of the test output:

```
---- tests::failing_test stdout ----
is 2 + 2 = 5?
thread 'tests::failing_test' panicked at 'assertion failed: `(left == right)`
  left: `4`,
 right: `5`', src/lib.rs:12:9
note: Run with `RUST_BACKTRACE=1` environment variable to display a backtrace.
```

When run with `cargo test -- --nocapture` instead, also the output of the passing test is shown:

```
running 2 tests
is 2 + 2 = 5?
is 2 + 2 = 4?
test tests::passing_test ... ok
test tests::failing_test ... FAILED
```

To avoid that the test output and test results are interleaved, restricting the test execution to one thread serializes the test execution and hence the output:

```
$ cargo test -- --nocapture --test-threads=1

running 2 tests
test tests::failing_test ... is 2 + 2 = 5?
FAILED
test tests::passing_test ... is 2 + 2 = 4?
ok
```

11.4.4 Test Selection

A subset of the available test cases can be run by indicating an expression to be matched by the names of the test functions to be executed. Given this module:

```
pub fn add(a: i32, b: i32) -> i32 {
    a + b
}

#[cfg(test)]
mod tests {
    use super::add;

    #[test]
    fn add_positive_to_positive() {
        assert_eq!(7, add(3, 4))
    }

    #[test]
    fn add_negative_to_positive() {
        assert_eq!(2, add(-4, 6))
    }

    #[test]
    fn add_negative_to_negative() {
        assert_eq!(-9, add(-2, -7))
    }
}
```

The expression `add` will match all three test functions:

```
$ cargo test add

running 3 tests
test tests::add_negative_to_negative ... ok
test tests::add_positive_to_positive ... ok
test tests::add_negative_to_positive ... ok

test result: ok. 3 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out
```

Whereas the expression `add_neg` will only match two of the three test cases, filtering out the third one:

```
$ cargo test add_neg

running 2 tests
test tests::add_negative_to_negative ... ok
test tests::add_negative_to_positive ... ok

test result: ok. 2 passed; 0 failed; 0 ignored; 0 measured; 1 filtered out
```

11.4.5 Ignoring Tests

A test case annotated with the `#[ignore]` attribute won't be run by default:

```
#[test]
#[ignore]
fn add_big_to_big() {
    assert_eq!(1000000, add(999999, 1))
}
```

```
$ cargo test

running 4 tests
test tests::add_big_to_big ... ignored
test tests::add_negative_to_negative ... ok
test tests::add_positive_to_positive ... ok
test tests::add_negative_to_positive ... ok

test result: ok. 3 passed; 0 failed; 1 ignored; 0 measured; 0 filtered out
```

Ignored tests can be run separately by setting the `--ignored` flag:

```
$ cargo test -- --ignored

running 1 test
test tests::add_big_to_big ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 3 filtered out
```

11.5 Test Organization

11.5.1 Unit Tests

Unit tests in Rust are written in a sub-module called `tests` and only test their super-module, not code of any other modules. The privacy rules of Rust allow a sub-module to call the private functions of its super-module, and hence, in contrast to many other test frameworks and programming languages, unit tests in Rust can also cover the private functions of a module. For unit tests, the test module must be annotated with `#[cfg(test)]`.

This module contains public functions for addition and multiplication, and a `tests` sub-module testing both the public and private functions (`calc/src/lib.rs`):

```
pub fn add(a: i32, b: i32) -> i32 {
    a + b
}

pub fn multiply(a: i32, b: i32) -> i32 {
    add_multiple_times(a, b)
}

fn add_multiple_times(a: i32, n: i32) -> i32 {
    let mut product = 0;
    for _i in 0..n {
        product += a;
    }
    product
}

#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    fn test_add() {
        assert_eq!(add(2, 3), 5);
    }

    #[test]
    fn test_multiply() {
        assert_eq!(multiply(3, 4), 12);
    }

    #[test]
    fn test_add_multuple_times() {
        assert_eq!(add_multiple_times(3, 2), 6);
    }
}
```

The multiplication is implemented as a repeated addition, and the unit test is able to cover both the public and private API.

11.5.2 Integration Tests

Integration tests in Rust only cover the public interface of the module to be tested. The test code is organized in a folder called `tests`, located next to the `src` folder. Since the integration test is neither part of the module to be tested, nor a sub-module thereof, the module to be tested must be integrated as an external crate (`calc/tests/calc_test.rs`):

```
extern crate calc;

#[test]
```

```rust
fn test_add() {
    assert_eq!(calc::add(2, 3), 5);
}

#[test]
fn test_multiply() {
    assert_eq!(calc::multiply(3, 4), 12);
}
```

The `#[cfg(test)]` annotation is *not* needed here. The integration test only covers the public API and thus simulates the usage of the module by another project.

Integration tests can be executed separately from unit tests by indicating the test files (without `.rs` suffix) to be executed:

```
$ cargo test --test calc_test
```

Each file in the `tests` folder is compiled as its own separate crate. Common functionality can be extracted and put into a module, for example called `common`.

This test sub-module sets up a test case for the addition test (`tests/common.rs`):

```rust
pub struct AdditionTest {
    pub a: i32,
    pub b: i32,
    pub expected: i32,
}

pub fn get_add_test_case() -> AdditionTest {
    AdditionTest {
        a: 3,
        b: 5,
        expected: 8,
    }
}
```

It can be used like this (`tests/calc_test.rs`):

```rust
extern crate calc;

mod common;

#[test]
fn test_add() {
    let test_case = common::get_add_test_case();
    assert_eq!(calc::add(test_case.a, test_case.b), test_case.expected);
}
```

If the module is defined in `tests/common.rs`, the module will be treated as an additional integration test (just without any test methods), but not so if the module is defined in `tests/common/mod.rs`.

11.5.3 Binary Crates

The functions in the `src/main.rs` file (of a binary crate) cannot be covered by integration tests. However, functions living in `src/lib.rs` can be both used by the code in `src/main.rs` and covered by unit and integration tests. Therefore, the code in `src/main.rs` should be kept small, moving as much of it to modules at possible, so that it can be automatically tested.

For example, the `main` function of the `calculator` binary crate only calls the `add` function from `src/lib.rs`:

```
extern crate calculator;

fn main() {
    println!("3 + 2 is {}", calculator::add(3, 2));
}
```

The file `src/lib.rs` contains both the `add` function and a test case for it:

```
pub fn add(a: i32, b: i32) -> i32 {
    return a + b;
}

#[cfg(test)]
mod tests {
    use super::add;

    #[test]
    fn test_add() {
        assert_eq!(add(2, 3), 5);
    }
}
```

The `add` function can also be tested using an integration test (`test/integration_tests.rs`):

```
extern crate calculator;

#[test]
fn test_add() {
    assert_eq!(calculator::add(2, 3), 5);
}
```

12 An I/O Project: Building a Command Line Program

Rust is fast, safe, works on different platforms, compiles to a single binary – and therefore is a good fit for command line tools.

The example application `minigrep` searches for a string in a file:

```
$ minigrep string file.txt
```

It is created as a binary crate:

```
$ cargo new --bin minigrep
```

In the Rust community, binary programs are structured as follows:

- The program is split up between `main.rs` and `lib.rs`.
- `main.rs` contains the command line parsing and some other configuration logic, as long as that logic is small.
- `lib.rs` contains the program's main logic, which is offered as a function called `run`.
- The `main` function calls `run` and deals with possible errors returned from there.

12.1 Command Line Arguments

The standard library contains an iterator `std::env::args` over strings, containing the command line arguments as passed to the program. (Alternatively, `std::env::args_os` provides an iterator of `OsString` for dealing with invalid unicode input.)

```
use std::env;

fn main() {
    let args: Vec<String> = env::args().collect();
    println!("{:?}", args);
}
```

Including the parent `std::env` and referring to the iterator as `env::args` is a good compromise between concise and readable code.

The iterator's `collect()` method turns the elements of the iterator into a collection, which must be specified using a type annotation.

The first command line argument is always the path to the binary being invoked:

```
$ cargo run query file.txt
["target/debug/minigrep", "query", "file.txt"]
```

The arguments of interest can be stored in variables for later use:

```rust
use std::env;

fn main() {
    let args: Vec<String> = env::args().collect();

    let query = &args[1];
    let filename = &args[2];

    println!("Searching for {}", query);
    println!("In file {}", filename);
}
```

```
$ cargo run query file.txt
Searching for query
In file file.txt
```

12.1.1 Extracting the Parsing Logic

The parsing logic is better split up, especially as it grows with further configuration parameters. A good approach is to extract these parts from `main`:

- a struct, containing all the configuration field
- a function, parsing the initial arguments into such a struct

The latter function can be implemented as a constructor of the struct:

```rust
use std::env;

fn main() {
    let args: Vec<String> = env::args().collect();
    let config = Config::new(&args);
    println!("Searching for {}", config.query);
    println!("In file {}", config.filename);
}

struct Config {
    query: String,
    filename: String,
}
```

```
impl Config {
    fn new(args: &[String]) -> Config {
        let query = args[1].clone();
        let filename = args[2].clone();

        Config { query, filename }
    }
}
```

Cloning strings, as in the constructor of `Config`, is not very efficient, but easy to implement. If only few objects are cloned, and only at the start of the program like in the example above, cloning will not undermine the program's performance in a critical manner.

12.2 Reading and Processing a File

The content of a text file can be read into a string variable using the `std::io::File` class and the traits from `std::io::predlude`:

```
use std::fs::File;
use std::io::prelude::*;

fn main() {
    let filename = &args[2]; // as above

    let mut f = File::open(filename).expect("file not found");
    let mut contents = String::new();
    f.read_to_string(&mut contents).expect("error reading file");

    println!("content:\n{}", contents)
}
```

12.3 Error Handling

This constructor from above causes a panic if too few arguments are provided:

```
impl Config {
    fn new(args: &[String]) -> Config {
        let query = args[1].clone();
        let filename = args[2].clone();

        Config { query, filename }
    }
}
```

If only one argument is provided, args has no element at index 2:

```
$ cargo run query
thread 'main' panicked at 'index out of bounds: the len is 2 but the index is 2'
```

12.3.1 User-Friendly Error Messages

This error message is intended for programmers, not for users. This panic message is a bit more user-friendly:

```
impl Config {
    fn new(args: &[String]) -> Config {
        if args.len() < 3 {
            panic!("not enough arguments");
        }
        // omitted
    }
}
```

```
$ cargo run query
thread 'main' panicked at 'not enough arguments'
```

12.3.2 Error Message Instead of Panic

It is better, though, to let the `main` function decide what to do in case of an error, than to shutdown the program with a panic right away. This version of the constructor returns a `Result` with either the parsed `Config`, or an error message if the parsing logic fails:

```
impl Config {
    fn new(args: &[String]) -> Result<Config, &'static str> {
        if args.len() < 3 {
            return Err("not enough arguments");
        }

        let query = args[1].clone();
        let filename = args[2].clone();

        Ok(Config { query, filename })
    }
}
```

The actual result as to be wrapped in the Ok or the Err variant, respectively. At the caller's side, the Result can be handled using the match keyword – or by the unwrap_or_else method of Result, which accepts a closure for the error message:

```rust
use std::env;
use std::process;

fn main() {
    let args: Vec<String> = env::args().collect();

    let config = Config::new(&args).unwrap_or_else(|err| {
        println!("Problem parsing arguments: {}", err);
        process::exit(1);
    });
}
```

The program is terminated with exit code 1 in case of a failure using the process::exit function. The error message is sent to the standard output:

```
$ cargo run query
Problem parsing arguments: not enough arguments
```

12.4 Refactoring

Once the command line parameters are wrapped up in a handy config object, the program logic can be extracted from the main function and put into a run function:

```rust
fn main() {
    let args: Vec<String> = env::args().collect();

    let config = Config::new(&args).unwrap_or_else(|err| {
        println!("Problem parsing arguments: {}", err);
        process::exit(1);
    });

    println!("Searching for {}", config.query);
    println!("In file {}", config.filename);

    run(config);
}

fn run(config: Config) {
    let mut f = File::open(config.filename).expect("file not found");
    let mut contents = String::new();
    f.read_to_string(&mut contents).expect("error reading file");
    println!("content:\n{}", contents)
}
```

The run function causes panics, which takes away control from the main function. To give that control back, this run implementation returns a result, wrapping possible errors up in a Box. Instead of calling except on the expressions that return a Result by themselves, the ? operator can be used:

```
use std::error::Error;

fn run(config: Config) -> Result<(), Box<Error>> {
    let mut f = File::open(config.filename)?;
    let mut contents = String::new();
    f.read_to_string(&mut contents)?;
    println!("content:\n{}", contents);
    Ok(())
}
```

The Ok variant only wraps the unit value (). In main, the error message can be used accordingly:

```
fn main() {
    // omitted

    if let Err(e) = run(config) {
        println!("Application error: {}", e);
        process::exit(1);
    }
}
```

Unlike Config::new, run does not a value that can be unwrapped. So here the if let construct is used to handle possible errors.

The extracted parts – the run function and the Config struct with its new method – can now be moved into a separated library crate, which makes it easier to reuse and test that code:

```
use std::error::Error;
use std::fs::File;
use std::io::prelude::*;

pub struct Config {
    pub query: String,
    pub filename: String,
}

impl Config {
    // omitted
}

pub fn run(config: Config) -> Result<(), Box<Error>> {
    // omitted
}
```

The `Config` struct, its fields, its `new` method, and the `run` function need to be public, so that they can be used from the stripped-down `main.rs`:

```
extern crate minigrep;

use std::env;
use std::process;

use minigrep::Config;

fn main() {
    let args: Vec<String> = env::args().collect();

    let config = Config::new(&args).unwrap_or_else(|err| {
        println!("Problem parsing arguments: {}", err);
        process::exit(1);
    });

    println!("Searching for {}", config.query);
    println!("In file {}", config.filename);

    if let Err(e) = minigrep::run(config) {
        println!("Application error: {}", e);
        process::exit(1);
    }
}
```

The extracted functionality now has to be included as an extern crate `minigrep`.

12.5 Test-Driven Development

The program will be finished from here in a process called Test-Driven Development (TDD), which works as follows:

1. Write test that fails for the expected reason.
2. Make the test compile and then pass.
3. Refactor the code without breaking the test.

4. Repeat from step 1 for the next part.

12.5.1 Failing Test Case

First, a (failing) test case – `one_result` – is added to `lib.rs`:

```
#[cfg(test)]
mod test {
    use super::*;

    #[test]
    fn one_result() {
        let query = "duct";
        let contents = "\
Rust:
safe, fast, productive.
Pick three.";

        assert_eq!(vec!["safe, fast, productive."], search(query, contents));
    }
}
```

12.5.2 Dummy Implementation

This code does not even compile, because the `search` function is missing. A dummy implementation, returning an empty vector, makes the code compile, but still lets the test fail:

```
pub fn search<'a>(query: &str, contents: &'a str) -> Vec<&'a str> {
    vec![]
}
```

The `search` function needs to annotate a lifetime: The strings in the resulting vectors are pulled out of the `contents` parameter, and therefore must life as long as that underlying string object.

12.5.3 Real Implementation

With this implementation of `search`, the test will pass:

```
pub fn search<'a>(query: &str, contents: &'a str) -> Vec<&'a str> {
    let mut results = Vec::new();

    for line in contents.lines() {
        if line.contains(query) {
            results.push(line);
        }
    }

    results
}
```

The text is analyzed line by line using the `lines` method of the `contents` string. The `contains` method returns true, if the given substring `query` is part of the string (the current line, that is). Matching lines are added to the `results` vector, which is returned from the function.

The test is now passing, but the program does not make use of `search` yet, which can be done as such:

```
pub fn run(config: Config) -> Result<(), Box<Error>> {
    let mut f = File::open(config.filename)?;
    let mut contents = String::new();
    f.read_to_string(&mut contents)?;
    for line in search(&config.query, &contents) {
        println!("{}", line);
    }
    Ok(())
}
```

The program now yields only the matching lines:

```
$ cargo run nobody poem.txt
Searching for nobody
In file poem.txt
I'm nobody! Who are you?
Are you nobody, too?
```

12.6 Environment Variable for Case Insensitive Search

`minigrep` needs an option to search through files in a case-insensitive manner. To make it possible for the user to make that option permanent instead of providing it with every invocation of the program, an environment variable should be introduced instead of a command line argument.

Continuing the test-driven approach from before, an additional (failing) test is added to the test module of `lib.rs`:

```
#[test]
fn case_insensitive() {
    let query = "rUsT";
    let contents = "\
Rust:
safe, fast, productive.
Pick three.
Trust me.";

    assert_eq!(
        vec!["Rust:", "Trust me."],
        search_case_insensitive(query, contents)
    );
}
```

For the implementation, both the query and every line to be tested are converted to lowercase:

```
pub fn search_case_insensitive<'a>(query: &str, contents: &'a str) -> Vec<&'a str> {
    let query = query.to_lowercase();
    let mut results = Vec::new();

    for line in contents.lines() {
        if line.to_lowercase().contains(&query) {
            results.push(line);
        }
    }

    results
}
```

The slice `query` is converted to a `String` by the `to_lowercase` method. Therefore, it has to be passed as `&query` into the `contains` method.

Both test cases now run successfully.

12.6.1 Additional Option

The new case insensitive option will be held by the `Config` struct:

```
pub struct Config {
    pub query: String,
    pub filename: String,
    pub case_sensitive: bool,
}
```

Depending on its value, either the `search` (case sensitive) or the `search_case_insensitive` function will be called from run:

```
pub fn run(config: Config) -> Result<(), Box<Error>> {
    let mut f = File::open(config.filename)?;
    let mut contents = String::new();
    f.read_to_string(&mut contents)?;

    let results = if config.case_sensitive {
        search(&config.query, &contents)
    } else {
        search_case_insensitive(&config.query, &contents)
    };

    for line in results {
        println!("{}", line);
    }
    Ok(())
}
```

If the environment variable `CASE_INSENSITIVE` is set (to any value), the according option should be set to `true`. The `var` function of the `env` module retrieves the value of the given environment variable and returns a `Result`. This logic belongs to the constructor of `Config`:

```
use std::env;

impl Config {
    pub fn new(args: &[String]) -> Result<Config, &'static str> {
        if args.len() < 3 {
            return Err("not enough arguments");
        }

        let query = args[1].clone();
        let filename = args[2].clone();

        let case_sensitive = env::var("CASE_INSENSITIVE").is_err();

        Ok(Config {
            query,
            filename,
            case_sensitive,
        })
    }
}
```

The `is_err` method returns true if the environment variable `CASE_INSENSITIVE` is not set, in which case a case-sensitive search has to be performed. In all other cases, a case-insensitive search was chosen by the user.

The behaviour of the program now can be changed by setting the `CASE_INSENSITIVE` environment variable:

```
$ cargo run To poem.txt
Searching for To
In file poem.txt
To tell your name the livelong day
To an admiring bog!

$ CASE_INSENSITIVE=1 cargo run To poem.txt
Searching for To
In file poem.txt
Are you nobody, too?
How dreary to be somebody!
To tell your name the livelong day
To an admiring bog!
```

The second invocation yields two additional lines, which matches the specified behaviour.

12.6.2 Printing to Standard Error

The program prints debug and error messages to standard output (`stdout`), just like the matching lines. In order to process the desired output – the matching lines – further with an other program, or to store it in a file, the debug and error messages should be sent to standard error (`stderr`) instead.

The `eprintln!` macro works just like the `println!` macro, but it prints to `stderr` instead of `stdout`. Since only the `main` function prints debug and error messages, it is the only place due for refactoring:

```rust
fn main() {
    let args: Vec<String> = env::args().collect();

    let config = Config::new(&args).unwrap_or_else(|err| {
        eprintln!("Problem parsing arguments: {}", err);
        process::exit(1);
    });

    eprintln!("Searching for {}", config.query);
    eprintln!("In file {}", config.filename);

    if let Err(e) = minigrep::run(config) {
        eprintln!("Application error: {}", e);
        process::exit(1);
    }
}
```

The debug and error messages can now be disposed of by forwarding them do `/dev/null`, for example:

```
$ cargo run To poem.txt 2> /dev/null
To tell your name the livelong day
To an admiring bog!

$ CASE_INSENSITIVE=1 cargo run To poem.txt 2> /dev/null
Are you nobody, too?
How dreary to be somebody!
To tell your name the livelong day
To an admiring bog!
```

13 Functional Language Features

Rust is heavily influenced by *functional programming*, which involves using functions as values.

13.1 Closures

A *closure* is a function-like construct that can be stored in a variable. Those are anonymous functions that are able to capture values from the surrounding scope. One common use case of closures is to abstract behaviour.

13.1.1 Closure Syntax

The following closure, bound to the name `add`, adds up two numbers and returns the result:

```rust
fn main() {
    let a = 3;
    let b = 5;
    let add = |x, y| x + y;
    let c = add(a, b);
    println!("{}", c);
}
```

Expressed as a function, `add` could be implemented as follows:

```
fn main() {
    let a = 3;
    let b = 5;
    fn add(x: u32, y: u32) -> u32 {
        x + y
    };
    let c = add(a, b);
    println!("{}", c);
}
```

Thus, the syntactical differences between a function and a closure are:

1. A function is declared using the `fn` keyword (expression), whereas a closure is usually bound to a name using the `let` keyword (statement).
2. A function's parameter list is surrounded by parentheses; closures use a pair of pipes (|) to surround the parameter list.
3. A function needs explicit type annotations for both the parameters and the return values. For closures, which usually are only relevant in a very limited scope, the compiler is able to figure out those types.
4. A function body needs to be surrounded by curly braces. For a closure with a body consisting of only a single expression, the curly braces can be left away.

The following declarations demostrate those differences:

```
fn  add1   (x: u32) -> u32 { x + 1 }  // 1. fn keyword, parentheses
let add2 = |x: u32| -> u32 { x + 1 }; // 2. let keyword, pipes
let add3 = |x|             { x + 1 }; // 3. no type annotations
let add4 = |x|               x + 1  ; // 4. no curly braces
```

13.1.2 Closure Semantics

Closures are useful to store some code to be executed later, possibly in a different context. This program declares the four basic arithmetic operations as closures and invokes them:

```
fn main() {
    let a = 16;
    let b = 4;

    let add = |x, y| x + y;
    let sub = |x, y| x - y;
    let mul = |x, y| x * y;
    let div = |x, y| x / y;

    println!("{}", add(a, b));
    println!("{}", sub(a, b));
    println!("{}", mul(a, b));
    println!("{}", div(a, b));
}
```

As opposed to a function, a closure can capture variables from its environment:

```
fn main() {
    let a = 7;
    let add = |x| a + x; // capture binding a

    let z = 3;
    println!("{}", add(z)); // 7 + 3 = 10
}
```

The compiler infers the types of a closure the first time it is used. Therefore, the same closure cannot be used twice with different types:

```
fn main() {
    let take_and_give = |value| value;

    let s = take_and_give(String::from("hello"));
    let n = take_and_give(5);
}
```

For the first call of `take_and_give`, the type `String` is inferred for the type of the `value` parameter. The second call, which uses an integer instead of a String, is therefore illegal:

```
error[E0308]: mismatched types
 --> src/main.rs:5:27
  |
5 |     let n = take_and_give(5);
  |                           ^
  |                           |
  |                           expected struct `std::string::String`, found integer
  |                           help: try using a conversion method: `5.to_string()`
  |
  = note: expected type `std::string::String`
             found type `{integer}`
```

13.1.3 Use-Case: Memoization

For expansive calculations, caching the results of an operation can safe a lot of execution time. This closure computes the `n`th fibonacci number iteratively, because closures cannot call themselves:

```rust
fn main() {
    let fib = |n| {
        if n < 2 {
            1
        } else {
            let mut prev1 = 0;
            let mut prev2 = 1;
            for _i in 0..n {
                let tmp = prev1 + prev2;
                prev1 = prev2;
                prev2 = tmp;
            }
            prev2
        }
    };
    for i in 0..10 {
        println!("fib({})={}", i, fib(i));
    }
}
```

Even though the recursive approach would be much heavier than this iterative implementation, caching the results in the program still illustrates the point.

First, a struct shall be defined, which holds both the closure and the calculated value:

```rust
struct Cache<T>
where
    T: Fn(u32) -> u32,
{
    operation: T,
    value: Option<u32>,
}
```

The struct `Cache` has a type parameter that is satisfied by any `Fn` mapping a `u32` to another `u32` – exactly what the `fib` closure does.

`Fn` is a trait provided by the standard library, which indicates that the function or closure at hand borrows values from its environment immutably. (The trait `FnMut` borrows values mutably; the trait `FnOnce` consumes values, and therefore can only be called once.)

The `value` field is declared as an `Option`, indicating that a value might already have been calculated – or not.

The `operation` and `value` fields are *not* public, for the client should not access them directly, but through a constructor and an additional method:

```
mpl<T> Cache<T>
where
    T: Fn(u32) -> u32,
{
    fn new(operation: T) -> Cache<T> {
        Cache {
            operation,
            value: None,
        }
    }

    fn value(&mut self, arg: u32) -> u32 {
        match self.value {
            Some(v) => v,
            None => {
                let v = (self.operation)(arg);
                self.value = Some(v);
                v
            }
        }
    }
}
```

The `new` function (the constructor) creates a new `Cache` with the given operation and no value. The `value` method checks if there is already a value, and returns it, if so. If not, `operation` is executed, and its result stored in the `value` field and returned to the caller.

The `Cache` can then be used as follows:

```
fn main() {
    let fib = |n| {
        println!("called fib closure");
        if n < 2 {
            1
        } else {
            let mut prev1 = 0;
            let mut prev2 = 1;
            for _i in 0..n {
                let tmp = prev1 + prev2;
                prev1 = prev2;
                prev2 = tmp;
            }
            prev2
        }
    };
    let mut cache = Cache::new(fib);
    println!("{}", cache.value(9));
    println!("{}", cache.value(9));
}
```

The `fib` closure was slightly modified to print a message when it is invoked. The 9th fibonacci number is requested two times, but the program only prints that additional message once, because the result of the second request was retrieved from the cache, and not calculated:

```
$ cargo run
called fib closure
55
55
```

13.1.4 Fixing the Cache Using a Map

This cache only works if it is used with the same parameter repeatedly. A useful implementation would not only store one result, but map the input parameter to cached results:

```
use std::collections::HashMap;

struct Cache<T>
where
    T: Fn(u32) -> u32,
{
    operation: T,
    values: HashMap<u32, u32>,
}

impl<T> Cache<T>
where
    T: Fn(u32) -> u32,
{
    fn new(operation: T) -> Cache<T> {
        Cache {
            operation,
            values: HashMap::new(),
        }
    }

    fn value(&mut self, arg: u32) -> u32 {
        match self.values.get(&arg) {
            Some(v) => *v,
            None => {
                let v = (self.operation)(arg);
                self.values.insert(arg, v);
                v
            }
        }
    }
}
```

The following client code now works as expected:

```
let mut cache = Cache::new(fib);
println!("{}", cache.value(9));
println!("{}", cache.value(10));
println!("{}", cache.value(9));
println!("{}", cache.value(10));
```

The `fib` closure is only called for the first time a parameter is used:

```
$ cargo run
called fib closure
55
called fib closure
89
55
89
```

13.2 Iterators

An iterator allows to process its underlying items one by one. The iterator takes care of the logic of iterating over the items, and determines when the sequence has reached its end.

An iterator can be retrieved by calling the `iter` method on a collection:

```
let v = vec![1, 2, 3];
let i = v.iter();
```

The iterator's elements then can be iterated over using a `for`/`in` loop:

```
for e in i {
    println!("{}", e);
}
```

13.2.1 The `Iterator` Trait and the `next` Method

An iterator must implement the `Iterator` trait, which is defined as follows:

```
pub trait Iterator {
    type Item;
    fn next(&mut self) -> Option<Self::Item>;
    // additional methods
}
```

The declaration `type Item` and the `Options` type `Self::Item` define a *associated type*. Implementing an iterator also requires defining an `Item` type.

The `next` method is the only method that implementors need to define. It moves on to the next item, and returns it wrapped up in a `Option`. The `self` reference is mutable, because calling `next` modifies the iterator's underlying state – the current position in the iterator.

An iterator can not only be processed using a `for/in` loop, but also directly using the `next` method. When doing so, the iterator reference has to be mutable explicitly. (The `for/in` loop makes the iterator mutable in the background.)

```
let v = vec![1, 2, 3];
let mut i = v.iter();
println!("{:?}", i.next());
println!("{:?}", i.next());
println!("{:?}", i.next());
println!("{:?}", i.next());
```

Output:

```
Some(1)
Some(2)
Some(3)
None
```

13.2.2 Consuming Adaptors

Methods that call the `next` method use up the iterator, and therefore are called _consuming adap-`tors. Thesum`` method is an example for this:

```
let v = vec![1, 2, 3];
let i = v.iter();
let total: i32 = i.sum();
println!("{}", total); // 6
```

The iterator `i` must not be used after calling the `sum` method. A `value used after move` compiler error would be the consequence of doing so nonetheless.

13.2.3 Iterator Adaptors

Methods that turn an iterator into another iterator are called *iterator adaptors*. Iterators are *lazu evaluated*: A consuming adaptor has to be applied in order to get the result of calling an iterator adaptor. For example, the `map` method does not really create a new iterator:

```
let v = vec![1, 2, 3];
v.iter().map(|x| x + 1);
```

Calling the `collect` method – a *consuming adaptor* – will produce a new collection with the mapped elements (here: added one to them):

```
let v = vec![1, 2, 3];
let plus_one: Vec<_> = v.iter().map(|x| x + 1).collect();
println!("{:?}", plus_one); // [2, 3, 4]
```

13.2.4 Filtering

The `filter` method takes a closure returning a boolean that is called for each of the iterator's items in turn. If the closure returns `true`, the item is included in the newly produces iterator; otherwise not.

```
let v = vec![1, 2, 3, 4, 5];
let even: Vec<_> = v.into_iter().filter(|x| x % 2 == 0).collect();
println!("{:?}", even); // [2, 4]
```

Notice that here the `into_iter` method is called, which takes ownership of the underlying collection. (Calling `iter_mut` would borrow the underlying elements mutably.)

13.2.5 Implementing an Iterator

To implement an own iterator, only the `next` method has to be implemented. A counter that runs from `0` to an arbitrary `limit` is defined as follows:

```
struct Counter {
    count: u32,
    limit: u32,
}

impl Counter {
    fn new(limit: u32) -> Counter {
        Counter {
            count: 0,
            limit: limit,
        }
    }
}
```

The iterator trait can be implemented as follows:

```
impl Iterator for Counter {
    type Item = u32;

    fn next(&mut self) -> Option<Self::Item> {
        self.count += 1;

        if self.count <= self.limit {
            Some(self.count)
        } else {
            None
        }
    }
}
```

The elements type is defined as `u32` using a type association. The `next` method increases its internal counter by one. If the counter is still within the limit, it is returned wrapped in the `Some` variant as an `Option`. Otherwise, the `None` variant is returned.

The counter can be used as follows:

```
let c = Counter::new(3);
for e in c {
    println!("{}", e);
}
```

This counter now offers different useful iterator methods by default:

```
let c1to5 = Counter::new(5); // [1, 2, 3, 4, 5]
let c1to7 = Counter::new(7); // [1, 2, 3, 4, 5, 6, 7]
let sum: u32 = c1to5
    .zip(c1to7.skip(2))      // [(1, 3), (2, 4), (3, 5), (4, 6), (5, 7)]
    .map(|(a, b)| a * b)     // [3, 8, 15, 24, 35]
    .filter(|x| x % 3 == 0)  // [3, 15, 24]
    .sum();
println!("{}", sum);         // 42
```

13.3 Improvements to `minigrep`

With knowledge about iterators, the code of the `minigrep` program from the last chapter can be improved.

13.3.1 Owning Iterator instead of `clone`

The `new` associated function of `Config` expects borrows the command line arguments in a slice of strings and clones them:

```rust
impl Config {
    pub fn new(args: &[String]) -> Result<Config, &'static str> {
        if args.len() < 3 {
            return Err("not enough arguments");
        }

        let query = args[1].clone();
        let filename = args[2].clone();

        let case_sensitive = env::var("CASE_INSENSITIVE").is_err();

        Ok(Config { query, filename, case_sensitive })
    }
}
```

At the moment, `env::args`, which is already an iterator, is converted to a vector in `main.rs`:

```rust
fn main() {
    let args: Vec<String> = env::args().collect();

    let config = Config::new(&args).unwrap_or_else(|err| {
        eprintln!("Problem parsing arguments: {}", err);
        process::exit(1);
    });

    // ...
}
```

This conversion using the `collect` method can be omitted, and `env::args` passed directly to the `Config::new` function:

```rust
fn main() {
    let config = Config::new(env::args()).unwrap_or_else(|err| {
        eprintln!("Problem parsing arguments: {}", err);
        process::exit(1);
    });

    // ...
}
```

This won't compile, because `Config::new` still expects a slice of strings. The parameter can be changed as follows, in order to expect the arguments as a mutable arguments iterator (see the documentation of `env::arg`):

```
impl Config {
    pub fn new(mut args: std::env::Args) -> Result<Config, &'static str> {

    // ...
    }
}
```

The logic to read the arguments is now rewritten in terms of iterators, i.e. the arguments are accessed one by one, handling the returned `Option` entries using the `match` keyword:

```
impl Config {
    pub fn new(mut args: std::env::Args) -> Result<Config, &'static str> {
        args.next();

        let query = match args.next() {
            Some(arg) => arg,
            None => return Err("Didn't get a query string"),
        };
        let filename = match args.next() {
            Some(arg) => arg,
            None => return Err("Didn't get a file name"),
        };

        // ...
    }
}
```

The first argument – the path to the executable – is passed. The next two arguments are interpreted as the query string and the filename parameters.

13.3.2 Iterator vs. Explicit Loop

The explicit loop over the content's lines in the `search` function uses an intermediary vector to hold the results:

```rust
pub fn search<'a>(query: &str, contents: &'a str) -> Vec<&'a str> {
    let mut results = Vec::new();

    for line in contents.lines() {
        if line.contains(query) {
            results.push(line);
        }
    }

    results
}
```

Iterators make it possible to write this code in a more concise way:

```rust
pub fn search<'a>(query: &str, contents: &'a str) -> Vec<&'a str> {
    contents
        .lines()
        .filter(|line| line.contains(query))
        .collect()
}
```

The `search_case_insensitive` function can be refactored using iterators, too:

```rust
pub fn search_case_insensitive<'a>(query: &str, contents: &'a str) -> Vec<&'a str> {
    let query = query.to_lowercase();
    contents
        .lines()
        .filter(|line| line.to_lowercase().contains(&query))
        .collect()
}
```

This code is more concise and focues on the function's logic instead of on the technical details – the *what* is emphasized, rather than the *how*.

Iterators are an example of Rust's *zero-cost abstractions*. This means, the more convenient iterator syntax is guaranteed to produce code without any overhead compared to the approach containing explicit loops and intermediary stores.

14 Cargo and Crates.io

The Cargo toolchain and the crate repository Crates.io form a strong ecosystem around the programming language. Cargo's documentation covers all the details.

14.1 Customizing Builds

A *release profile* is a set of configuration for code compilation. By default, there's a `dev` and a `release` profile. The `dev` profile is used when `cargo build` is invoked without any special flag. The `release` profile can be used when setting the `--release` flag. The profile used will be shown in the output:

```
$ cargo build
Finished dev [unoptimized + debuginfo] target(s) in 0.00s

$ cargo build --release
Finished release [optimized] target(s) in 0.00s
```

The details for every profile can be defined in `Cargo.toml`, by defining a section for the profile of interest:

```toml
[profile.dev]
opt-level = 1

[profile.release]
opt-level = 2
```

This configuration sets the optimization level of the `dev` profile to 1 (as opposed to the default of 0) and for the `release` profile to 2 (default: 3), just for the sake of demonstration.

Having the lowest optimization level for `dev` and the highest for `release` is a sensible configuration, for during development the build should be fast, but the execution time of the binary hardly matters, whereas in production the compilation time is hardly an issue, while execution speed of the resulting binary is critical.

14.2 Publishing Crates

Crates.io is not only a one-way channel to retrieve code from. It is also a place where developers can share their own code. In order to be useful to other developers, the code must provide useful documentation, accurate metadata, a useful API, and a comprehensive test-suite.

Those requirements are best demonstrated using an example: The library crate `mememo`, which is short for *mean, median, and mode*` , implements trivial implementations for the statistical concepts it is named after.

14.2.1 The `mememo` Crate

The `mememo` crate provides three public functions, which all expect a vector of integers (`i32`):

1. `mean`: Calculates the mean, i.e. sums up the elements of the vectors and divides the sum by the number of elements.
2. `median`: Returns the value of the middle element of the given vector.
3. `mode`: Returns the item that has the most occurrences in the given vector.

`mean` returns a `f64`, because a division of a sum is not to be expected being a discrete number. `mode` returns an integer, because the result of this operation is taken from the input vector. For `median`, there are two possible outcomes:

- If the number of elements in the input vector is *odd*, the middle element of the vector is the result, and thus the return type is the same as the input vector's element type: `i32`.
- If the number of elements in the input vector is *even*, the median is calculated as the mean of the two middle elements. In this case, the result is a `f64`.

These possible outcomes can be expressed using an enum with two variants:

```rust
pub enum Median {
    MiddleSingle(i32),
    MiddleTwoMean(f64),
}
```

The function headers of the three functions look as follows:

```rust
pub fn mean(numbers: &Vec<i32>) -> f64 {
    // ...
}

pub fn median(numbers: &Vec<i32>) -> Median {
    // ...
}

pub fn mode(numbers: &Vec<i32>) -> i32 {
    // ...
}
```

The implementations are omitted and can be seen in the file `code/mememo/src/lib.rs`.

14.2.2 Documentation Comments

Whereas ordinary comments clarify details on specific sections of code, *documentation comments* are intended for the user of the public API rather than for a maintainer of the code.

Rust code is commented using two slashes (//), and those comments can occur anywhere in the code. Documentation comments start with three slashes (///) and must be located just before the element they're documenting. Those comments should indicate how to use the public item at hand. For this purpose, Markdown syntax can be used, as well as code examples.

Here's the documentation comment of the `mean` function discussed above:

````
/// Calculates the mean of the elements in the given vector.
///
/// # Example
///
/// ```
/// let numbers = vec![1, 2, 3, 4];
/// assert_eq!(2.5, mememo::mean(&numbers));
/// ```
pub fn mean(numbers: &Vec<i32>) -> f64 {
    // ...
}
````

The `cargo doc` tool generates HTML documentation based on these comments and puts it into the `target/doc` folder. The documentation can be generated an opened in a browser by invoking `cargo doc --open`.

It is important that the code examples used in documentation comments do actually compile – and pass the assertions used. Broken example code is annoying, and `cargo test` makes sure, the example code is working code, by executing the code as `Doc-tests`:

```
$ cargo test

...

Doc-tests mememo
test src/lib.rs - mean (line 11) ... ok
test src/lib.rs - mode (line 65) ... ok
test src/lib.rs - median (line 33) ... ok
test src/lib.rs - median (line 44) ... ok

test result: ok. 4 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out
```

Having an *Example* section is common for Rust crates. Other sections commonly provided are:

- Panics: scenarios in which the code will panic (to be avoided by the caller)
- Errors: kinds of errors being returned and their underlying conditions
- Safety: invariants the caller needs to hold up when invoking `unsafe` code

In order to document the item that contains the comment, the comment style `//!` can be used. For example, the entire `mememo` crate (`src/lib.rs`) can be documented as follows:

```
//! # mememo
//!
//! The crate _mememo_ provides trivial implementations of the operations
//! _mean_, _median_, and _mode_. This crate is not intended for productive
//! use, but only to demonstrate the use of crates and other Rust features.

use std::collections::HashMap;

/// Calculates the mean of the ...
```

There's no code following this comment (the import section only starts after an empty line). There's another documentation comment below, which belongs to the `mean` function further down the file.

These kinds of comments appear on the front page of the crate's documentation.

14.2.3 Re-exports

The internal structure of the code often does not provide the most convenient API possible. The code examples of the `median` function serve as good example for the verbosity an API user is facing:

```
let numbers = vec![1, 2, 3, 4, 5];
match mememo::median(&numbers) {
    mememo::Median::MiddleSingle(got) => assert_eq!(3, got), // verbose!
    _ => panic!("wrong median calculation"),
}
let numbers = vec![1, 2, 3, 4];
match mememo::median(&numbers) {
    mememo::Median::MiddleTwoMean(got) => assert_eq!(2.5, got), // verbose!
    _ => panic!("wrong median calculation"),
};
```

It would be much more convenient if the client could write `mememo::MiddleSingle` or `mememo::MiddleTwoMean` instead of always having to add the `Median` in between.

Rust allows to re-export such items under a more convenient name using the `pub use` notation. The internal structure of the code is preserved, but the client has it much easier to deal with the exposed API.

The enum's variants can be re-exported as follows:

```rust
pub use Median::MiddleSingle;
pub use Median::MiddleTwoMean;
```

These re-exports are now displayed in an additional section (*Re-exports*) in the documentation. The API can now be used as follows:

```rust
let numbers = vec![1, 2, 3, 4, 5];
match mememo::median(&numbers) {
    mememo::MiddleSingle(got) => assert_eq!(3, got), // less verbose!
    _ => panic!("wrong median calculation"),
}
let numbers = vec![1, 2, 3, 4];
match mememo::median(&numbers) {
    mememo::MiddleTwoMean(got) => assert_eq!(2.5, got), // less verbose!
    _ => panic!("wrong median calculation"),
};
```

14.2.4 Crate Metadata

When a new crate is created using `cargo new`, some metadata is automatically put into `Cargo.toml`, such as the crate's name, the initial version, and the author's details (which are obtained from the `git` config). In order to publish a crate on Crates.io, at least two additional fields need to be defined:

1. `description`: a short description (one or two sentences) about the crate. This description will be displayed in the search results on Crates.io.
2. `license`: a *license identifier value* specifying the license the code is released under. Those identifiers can be obtained from the Linux Foundation's Software Package Data Exchange (SPDX) under spdx.org/licenses. Multiple licenses can be listed separated by `OR`.

For example, `Cargo.toml` of `mememo` with an added `description` and dual licensing (MIT and GPLv3 or later) looks as follows:

```toml
[package]
name = "mememo"
version = "0.1.0"
description = "mememo stands for Mean, Median, Mode. It provides trivial..."
authors = ["Patrick Bucher <patrick.bucher@stud.hslu.ch>"]
edition = "2018"
license = "MIT OR GPL-3.0-or-later"

[dependencies]
```

14.2.5 Publishing the Crate

In order to publish a crate, an account on Crates.io is needed, which currently requires a GitHub account. After the account is created, a new API key can be generated and obtained under crates.io/me. This key can be used to login to Crates.io from the local computer:

```
$ cargo login abcdefghijklmnopqrstuvwxyz012345
```

This key is stored locally under `~/.cargo/credentials` and is considered a *secret*, thus must not be shared with others. To logout, simply delete the key from the `credentials` file.

Before publishing code, it is import to consider that a publish is permanent. Existing code can be neither deleted nor owerwritten. It is only possible to publish additional code with a higher version number.

When the crate is ready for publishing – with a sufficient code-quality, without known critical bugs, with a useful and well-documented public API, a test-suite, and the necessary metadata – the crate can be published:

```
$ cargo publish
```

To publish a new (higher) version of an existing crate, simply increase the `version` metadata field in `Cargo.toml` according to the rules of Semantic Versioning (semver.org) and then call `cargo publish` again. It is a good idea to also tag the version in the SCM system, e.g. using `git tag v0.1.1` to add the version tag `v0.1.1` to current committed state of the repository.

14.2.6 Un-publishing Versions

Even though it is not possible to remove crates or specific versions thereof from Crates.io, it is possible to prevent future use of specific versions of the crate using the `cargo yan` command.

Let's say the version `0.1.0` is buggy, and therefore a later version `0.1.1` was released fixing those bugs, future use of the version `0.1.0` can be prevented as follows:

```
$ cargo yank --vers 0.1.0
```

If an existing project already uses that version and thus has an entry for it in `Cargo.lock`, the project still builds, and the version `0.1.0` can still be used from that project. New projects, however, are required to use the later version `0.1.1` as a dependency – unless the the yank operation is undone, allowing future use for the specified version again:

```
$ cargo yank --vers 0.1.0 --undo
```

14.3 Workspaces

Oftentimes, crates are not worked on in isolation, but together as a project. Cargo makes working on multiple packages belonging together more comfortable using *workspaces*. Commands like `cargo fmt` and `cargo test` are applied on all packages of the workspace, saving the developer a lot of typing and work.

A workspace shares the binary output and the `Cargo.lock` file; with the latter it is ensured that all packages belonging to the workspace use the same versions of the dependent crates.

Technically, a workspace is a folder containing a `Cargo.toml` file with a `[workspace]` section. Here, the `members` are listed, i.e. the packages belonging to the workspace.

14.3.1 Example: `mean` workspace

For the sake of an example, the workspace `mean` for a trivial project should be created. It contains two library crates `sum` and `avg`, and one binary crate `mean`; the latter depending on the former two.

First, the workspace directory is created and entered:

```
$ mkdir mean
$ cd mean
```

Second, the library crates `sum` and `avg`, as well as the binary crate `mean` are created:

```
$ cargo new --lib sum
$ cargo new --lib avg
$ cargo new --bin mean
```

The directory structure now looks like this:

```
mean
├── Cargo.lock
├── Cargo.toml
├── avg
│   ├── Cargo.toml
│   └── src
│       └── lib.rs
├── mean
│   ├── Cargo.toml
│   └── src
│       └── main.rs
```

```
└── sum
    ├── Cargo.toml
    └── src
        └── lib.rs
```

Third, the `Config.toml` file is created, declaring the crates just created as members of the workspace:

```
[workspace]

members = [
    "sum", "avg", "mean"
]
```

The workspace can be build using `cargo build`, putting the resulting binaries in the shared `target` directory, which helps avoiding unnecessary rebuilding operations.

For the two library crates `sum` and `avg`, one public function has to implemented for each.

The `sum` library needs a `sum` function, which sums up a vector of integers and returns that sum (`sum/src/lib.rs`):

```
/// Sums up the given vector and returns the sum.
///
/// # Example
///
/// ```
/// let numbers = vec![1, 2, 3, 4];
/// assert_eq!(10, sum::sum(numbers));
/// ```
pub fn sum(numbers: Vec<i32>) -> i32 {
    let mut result = 0;
    for n in numbers {
        result += n;
    }
    result
}
```

The `avg` library needs a `avg` function, which calculates the average of a given vector of integers (`sum/src/lib.rs`):

```
extern crate sum;

/// Calculates the average of the given vector and returns it.
///
/// # Example
```

```rust
///
/// ```
/// let numbers = vec![1, 2, 3, 4];
/// assert_eq!(2.5, avg::avg(numbers));
/// ```
pub fn avg(numbers: Vec<i32>) -> f64 {
    let n = numbers.len();
    let sum = sum::sum(numbers);
    let average: f64 = (sum as f64) / (n as f64);
    average
}
```

This implementation makes use of the `sum` crate implemented just before, which needs to be declared as a dependency (`avg/Cargo.toml`):

```toml
[dependencies]

sum = { path = "../sum" }
```

The `mean` crate invokes the `avg` function and outputs both the input vector and result (`mean/src/main.rs`):

```rust
extern crate avg;

fn main() {
    let numbers = vec![1, 2, 3, 4, 5, 6, 7, 8, 9, 10];
    println!("numbers: {:?}", numbers);
    let average = avg::avg(numbers);
    println!("mean thereof: {}", average);
}
```

Again, the external crate needs to referred as a dependency (`mean/Cargo.toml`):

```toml
[dependencies]
avg = { path = "../avg" }
```

The program can be invoked using `cargo run`. If a workspace contains multiple binary crates, the crate to be run has to be specified using the `-p` flag:

```
$ cargo run -p mean
numbers: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
mean thereof: 5.5
```

Builds, tests, formatters and the like can also be run for single workspace members using the `-p` flag:

```
$ cargo build -p avg
$ cargo test -p sum
$ cargo fmt -p mean
```

Crates within a workspace all need to be published seperately. For version control, git submodules might be a good fit for organizing a workspace.

14.4 Cargo Binaries

Crates.io is not only a place to share code, it can also be used to publish programs. Using the `cargo install` command, the output of a binary crate can be installed locally. This command installs the popular `ripgrep` command:

```
$ cargo install ripgrep
```

The `ripgrep` crate and its dependencies are downloaded, built, and the resulting binary `ripgrep` is installed under `~/.cargo/bin/rg`. If the parent folder `~/.cargo/bin` is part of the `$PATH` variable, `rg` can be invoked directly:

```
$ rg --version
ripgrep 11.0.2
-SIMD -AVX (compiled)
+SIMD +AVX (runtime)
```

Binaries in the `~/.cargo/bin` folder named `cargo-[something]` are treated as cargo subcommands and can be invoked as `cargo [something]`. The command `cargo --list` shows all installed subcommands. For example, `cargo fmt` invokes the program under `~/.cargo/bin/cargo-fmt`.

Binary crates can be removed using `cargo uninstall`:

```
$ cargo uninstall ripgrep
```

15 Smart Pointers

15.1 `Box<T>`: Store Data on the Heap

15.1.1 Recursive Types: Cons List

15.2 `Deref` Trait

15.2.1 Custom Smart Pointers

15.2.2 Deref Coercion

15.3 `Drop` Trait

15.3.1 Dropping Early

15.4 `Rc<T>`: Count References

15.4.1 Share Data

15.5 `RefCell<T>`

15.5.1 Interior Mutability Pattern

15.5.2 Multiple Mutable Owners

15.6 Referency Cycles and Memory Leaks

15.6.1 `Weak<T>`: Prevent Reference Cycles