Rust: Fast & Safe

Jakub Beránek, Mathieu Fehr, Saurabh Raje

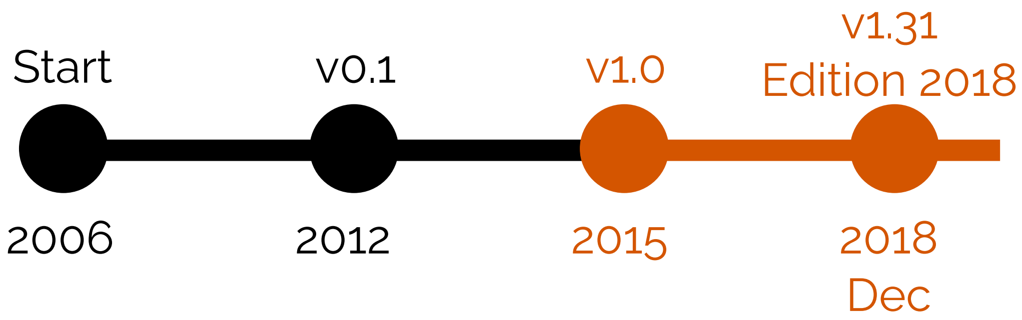

Our C++ app keeps crashing. How do we fix this?
Address sanitizer
More assertions
Rewrite it in Rust?
imgflip.com

What is Rust?

System programming language for building reliable and efficient software.

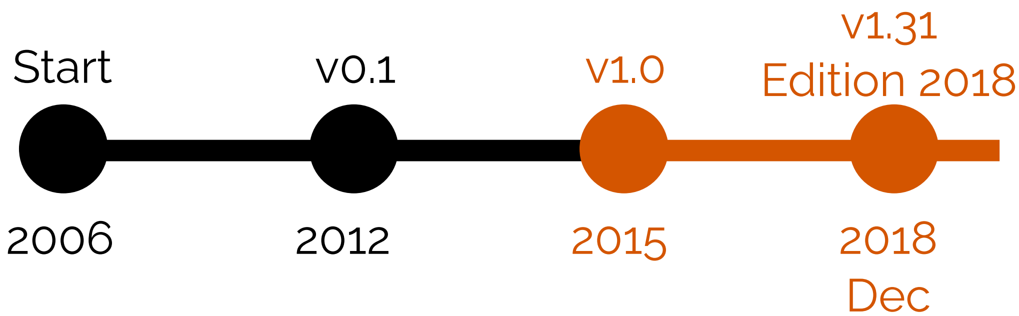
What is Rust?

System programming language for building reliable and efficient software.



What is Rust?

System programming language for building reliable and efficient software.



Fast & Safe

Fast & Safe

Quick development

Fast & Safe

Quick development

High performance

Fast & Safe

Quick development

High performance

Project management (Cargo)

Using libraries

Cargo.toml

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

[dependencies]
ibverbs = "0.4"
json = "1.0"
protobuf = "2.0"
```

Using libraries

Cargo.toml

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

[dependencies]
ibverbs = "0.4"
json = "1.0"
protobuf = "2.0"
```

main.rs

```rust
use json::parse;

fn main() {
    parse("data.json");
}
```

Using libraries

Cargo.toml

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

[dependencies]
ibverbs = "0.4"
json = "1.0"
protobuf = "2.0"
```

main.rs

```rust
use json::parse;

fn main() {
    parse("data.json");
}
```

More than 26k libraries available

Unified documentation

Struct ibverbs::CompletionQueue [−][src]

[+] Show declaration

[−] A completion queue that allows subscribing to the completion of queued sends and receives.

Methods

[−] `impl<'ctx> CompletionQueue<'ctx>` [src]

[−] [src]

```
pub fn poll<'c>(
    &self,
    completions: &'c mut [ibv_wc]
) -> Result<&'c mut [ibv_wc], ()>
```

Poll for (possibly multiple) work completions.

A Work Completion indicates that a Work Request in a Work Queue, and all of the outstanding unsignaled Work Requests that posted to that Work Queue, associated with this CQ have completed. Any Receive Requests, signaled Send Requests and Send Requests that ended with an error will generate Work Completions.

When a Work Request ends, a Work Completion is added to the tail of the CQ that this Work Queue is associated with. `poll` checks if Work Completions are present in a CQ, and pop them from the head of the CQ in the order they entered it (FIFO) into `completions`. After a Work Completion was popped from a CQ, it cannot be returned to it. `poll` returns the subset of `completions` that successfully completed. If the returned slice has fewer elements than the provided `completions` slice, the CQ was emptied.

Not all attributes of the completed `ibv_wc`'s are always valid. If the completion status is not `IBV_WC_SUCCESS`, only the following attributes are valid: `wr_id`, `status`, `qp_num`, and `vendor_err`.

Note that `poll` does not block or cause a context switch. This is why RDMA technologies can achieve very low latency (below 1 µs).

Integrated tooling

Build

```
$ cargo build
```

Integrated tooling

Build

Run

```
$ cargo build
$ cargo run
```

Integrated tooling (tests)

```rust
#[test]
fn test_add() {
    assert_eq!(add(1, 2), 3);
}
```

Integrated tooling (tests)

```rust
#[test]
fn test_add() {
    assert_eq!(add(1, 2), 3);
}
```

```
$ cargo test
```

Integrated tooling (benchmarks)

```rust
#[bench]
fn bench_add_two(b: &mut Bencher) {
    b.iter(|| add_two(2));
}
```

Integrated tooling (benchmarks)

```rust
#[bench]
fn bench_add_two(b: &mut Bencher) {
    b.iter(|| add_two(2));
}
```

```
$ cargo bench
```

Integrated tooling

Format

```
$ cargo fmt
```

Integrated tooling

Format

Lint

```
$ cargo fmt
$ cargo clippy
```

Integrated tooling

Format

Lint

Publish ~~to SC~~

```
$ cargo fmt
$ cargo clippy
$ cargo publish
```

Build scripts

build.rs

```
fn main() {
    // generate Protobuf objects
    protoc_rust::run("protobuf/message.proto", "src/protos");




}
```

Build scripts

build.rs

```
fn main() {
    // generate Protobuf objects
    protoc_rust::run("protobuf/message.proto", "src/protos");

    // generate C headers
    cbindgen::Builder::new()
      .generate()
      .write_to_file("bindings.h");
}
```

(interlude)

Multi-phase compiler

```rust
fn main() {
    look_ma_no_forward_declaration();
}

fn look_ma_no_forward_declaration() { }
```

Proper module system

foo.rs

```rust
pub fn fun1() {
    println!("fun1");
}
fn fun2() {
    println!("fun2");
}
```

Proper module system

foo.rs

```rust
pub fn fun1() {
    println!("fun1");
}
fn fun2() {
    println!("fun2");
}
```

main.rs

```rust
use foo;

fn main() {
    foo::fun1();
    // foo::fun2(); private
}
```

Proper module system

foo.rs

```
pub fn fun1() {
    println!("fun1");
}
fn fun2() {
    println!("fun2");
}
```

main.rs

```
use foo;

fn main() {
    foo::fun1();
    // foo::fun2(); private
}
```

- visibility control
- self-contained

Structures

```rust
struct Person {
    pub age: u32,
    name: String
}
```

Structures

```rust
struct Person {
    pub age: u32,
    name: String
}

impl Person {
    pub fn new(age: u32, name: String) -> Person {
        Person { age, name }
    }
    pub fn is_adult(&self) -> bool {
        self.age >= 18
    }
}
```

Traits

```
trait Buffer {
    fn size(&self) -> usize;
    fn read(&self) -> u8;
}
```

Traits

```rust
trait Buffer {
    fn size(&self) -> usize;
    fn read(&self) -> u8;
}
```

```rust
struct MemBuffer { data: Vec<u8> }
```

Traits

```
trait Buffer {
    fn size(&self) -> usize;
    fn read(&self) -> u8;
}
```

```
struct MemBuffer { data: Vec<u8> }

impl Buffer for MemBuffer {
    fn size(&self) -> usize { self.data.size() }
    fn read(&self) -> u8 { ... }
}
```

Traits

```rust
trait Buffer {
    fn size(&self) -> usize;
    fn read(&self) -> u8;
}
```

```rust
struct MemBuffer { data: Vec<u8> }

impl Buffer for MemBuffer {
    fn size(&self) -> usize { self.data.size() }
    fn read(&self) -> u8 { ... }
}
```

```rust
struct FileBuffer { path: String }
```

Traits

```rust
trait Buffer {
    fn size(&self) -> usize;
    fn read(&self) -> u8;
}
```

```rust
struct MemBuffer { data: Vec<u8> }

impl Buffer for MemBuffer {
    fn size(&self) -> usize { self.data.size() }
    fn read(&self) -> u8 { ... }
}
```

```rust
struct FileBuffer { path: String }

impl Buffer for FileBuffer {
    fn size() -> usize { fs::metadata(self.path).len() }
    fn read(&self) -> u8 { ... }
}
```

Built-in traits

```rust
impl Display for Person {
    fn fmt(&self, f: Formatter) -> Result { ... }
}
println!("{}", person);
```

Built-in traits

```
impl Display for Person {
    fn fmt(&self, f: Formatter) -> Result { ... }
}
println!("{}", person);
```

```
impl From<String> for IPAddress {
    fn from(value: String) -> IPAddress { ... }
}
let ip: IPAddress = "127.0.0.1".into();
```

Built-in traits

```rust
impl Display for Person {
    fn fmt(&self, f: Formatter) -> Result { ... }
}
println!("{}", person);
```

```rust
impl From<String> for IPAddress {
    fn from(value: String) -> IPAddress { ... }
}
let ip: IPAddress = "127.0.0.1".into();
```

```rust
impl Add for Matrix {
    fn add(self, other: Matrix) -> Matrix { ... }
}
let c: Matrix = matA + matB;
```

Generics

```
struct KeyValue<K, V> {
    key: K,
    value: V
}
```

Generics

```
struct KeyValue<K, V> {
    key: K,
    value: V
}

trait Buffer<T> {
    fn read(&self) -> T;
}
```

Generics

```
struct KeyValue<K, V> {
    key: K,
    value: V
}

trait Buffer<T> {
    fn read(&self) -> T;
}
```

```
fn print_buffer<B: Buffer<T>, T: Display>(buffer: B) {
    println!("{}", buffer.read());
}
```

Generics

```
struct KeyValue<K, V> {
    key: K,
    value: V
}

trait Buffer<T> {
    fn read(&self) -> T;
}
```

```
fn print_buffer<B: Buffer<T>, T: Display>(buffer: B) {
    println!("{}", buffer.read());
}
```

```
fn print_bigger<T: PartialEq + Display>(a: T, b: T) {
    if (a > b) { println!("{}", a); }
}
```

Generics

```
struct KeyValue<K, V> {
    key: K,
    value: V
}

trait Buffer<T> {
    fn read(&self) -> T;
}
```

```
fn print_buffer<B: Buffer<T>, T: Display>(buffer: B) {
    println!("{}", buffer.read());
}
```

```
fn print_bigger<T: PartialEq + Display>(a: T, b: T) {
    if (a > b) { println!("{}", a); }
}
```

```
impl <T: Display> Serialize for T {
    ...
}
```

Algebraic data types/tagged unions/sum types/ discriminated unions/variants

```
enum Packet {
    Header { source: u32, tag: u32, data: Vec<u8> },
    Payload { data: Vec<u8> },
    Ack { seq: u64 }
}
```

Algebraic data types/tagged unions/sum types/ discriminated unions/variants

```rust
enum Packet {
    Header { source: u32, tag: u32, data: Vec<u8> },
    Payload { data: Vec<u8> },
    Ack { seq: u64 }
}
```

Pattern matching

```rust
match socket.get_packet() {
    Header {data, ..} | Payload {data, ..} => { },
    _ => { println!("Packet without data"); }
}
```

Algebraic data types/tagged unions/sum types/ discriminated unions/variants

```
enum Packet {
    Header { source: u32, tag: u32, data: Vec<u8> },
    Payload { data: Vec<u8> },
    Ack { seq: u64 }
}
```

Pattern matching

```
match socket.get_packet() {
    Header {data, ..} | Payload {data, ..} => { },
    _ => { println!("Packet without data"); }
}
```

The compiler forces you to handle all variants

Error handling

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

Error handling

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

```rust
fn find_index(items: &[u32], item: u32) -> Option<usize> {
    ...
}
```

Error handling

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

```rust
fn find_index(items: &[u32], item: u32) -> Option<usize> {
    ...
}

let index = match find_index(&[1, 2, 3], 4) {
    Some(index) => println!("index found: {}", index),
    None => println!("index not found")
}
```

Error handling

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

Error handling

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

```
fn find_in_db(...) -> Result<Vec<DbRow>, DbError> {
    ...
}
```

Error handling

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

```
fn find_in_db(...) -> Result<Vec<DbRow>, DbError> {
    ...
}

let item = match find_in_db(...) {
    Ok(item) => item,
    Err(error) => {
        println!("Error: {}", error);
        vec!()
    }
}
```

Error handling

```rust
fn download(address: String) -> Result<Vec<u8>> {
    let ip = address.parse()?;
    let client = TcpStream::connect(ip)?;

    let mut buf = vec!();
    client.read(&mut buf)?;
    buf
}
```

Error handling

```rust
fn download(address: String) -> Result<Vec<u8>> {
    let ip = address.parse()?;
    let client = TcpStream::connect(ip)?;

    let mut buf = vec!();
    client.read(&mut buf)?;
    buf
}
```

```rust
let item = value?;
```

Error handling

```rust
fn download(address: String) -> Result<Vec<u8>> {
    let ip = address.parse()?;
    let client = TcpStream::connect(ip)?;

    let mut buf = vec!();
    client.read(&mut buf)?;
    buf
}
```

```rust
let item = value?;

// expands to
let item = match value {
    Ok(v) => v,
    Err(e) => return Err(e)
};
```

Macros

```
macro_rules! find_min {
    ($x:expr) => ($x);




}
```

Macros

```
macro_rules! find_min {
    ($x:expr) => ($x);
    ($x:expr, $($y:expr),+) => (
        std::cmp::min($x, find_min!($($y),+))
    )
}
```

Macros

```
macro_rules! find_min {
    ($x:expr) => ($x);
    ($x:expr, $($y:expr),+) => (
        std::cmp::min($x, find_min!($($y),+))
    )
}
```

```
find_min!(5);       // 5
find_min!(2, 1, 3); // 1
```

Macros

```
macro_rules! find_min {
    ($x:expr) => ($x);
    ($x:expr, $($y:expr),+) => (
        std::cmp::min($x, find_min!($($y),+))
    )
}
```

```
find_min!(5);       // 5
find_min!(2, 1, 3); // 1
```

```
macro_rules! create_function {
    ($func_name:ident) => (
        fn $func_name() {
            println!("You called {:?}()", stringify!($func_name));
        }
    )
}
```

Procedural macros

```rust
fn my_macro(attr: TokenStream, item: TokenStream) -> TokenStream {
    ...
}
```

Procedural macros

```
fn my_macro(attr: TokenStream, item: TokenStream) -> TokenStream {
    ...
}
```

```
#[derive(my_macro)]
struct Record {
    #[my_macro]
    pub id: u32
}
```

Procedural macros

```rust
#[derive(Serialize, Deserialize)]
struct Person { name: String, age: u32 }
```

Procedural macros

```
#[derive(Serialize, Deserialize)]
struct Person { name: String, age: u32 }

json::to_string(person);
yaml::to_string(person);
let person = json::parse(person_str);
```

Procedural macros

```rust
#[derive(Serialize, Deserialize)]
struct Person { name: String, age: u32 }

json::to_string(person);
yaml::to_string(person);
let person = json::parse(person_str);
```

```rust
#[derive(CmdArgs)]
struct Args {
    #[arg(short = "d", long = "debug")]
    debug: bool,
    #[arg(parse(from_os_str))]
    path: PathBuf,
}
```

Procedural macros

```
#[derive(Serialize, Deserialize)]
struct Person { name: String, age: u32 }

json::to_string(person);
yaml::to_string(person);
let person = json::parse(person_str);
```

```
#[derive(CmdArgs)]
struct Args {
    #[arg(short = "d", long = "debug")]
    debug: bool,
    #[arg(parse(from_os_str))]
    path: PathBuf,
}
```

```
#[derive(Debug)]
struct Person { name: String, age: u32 }

println!("{:?}", person);
```

Type inference

```
let elem = 5u8;
```

Type inference

```rust
let elem = 5u8;
let mut vec = Vec::new();
```

Type inference

```rust
let elem = 5u8;
let mut vec = Vec::new();
vec.push(elem);
```

Type inference

```rust
let elem = 5u8;
let mut vec = Vec::new();
vec.push(elem);
// vec is now Vec<u8>
```

Iterators

```
vec.iter()
```

Iterators

```
vec.iter()
    .zip(iter2)
```

Iterators

```
vec.iter()
    .zip(iter2)
    .filter(|(a, b)| a > b)
```

Iterators

```
vec.iter()
    .zip(iter2)
    .filter(|(a, b)| a > b)
    .map(|(a, b)| a * b)
```

Iterators

```
vec.iter()
    .zip(iter2)
    .filter(|(a, b)| a > b)
    .map(|(a, b)| a * b)
    .sum::<i32>();
```

Generators

```
let mut fibonacci = || {
    yield 1;

    let mut a = 0;
    let mut b = 1;
    loop {
        yield a + b;
        a = b;
        b = a + b;
    }
};
let f = fibonacci().iter().take(5).collect();
```

Async/await

```
async fn compute_job(job: Job) -> Result<Data, Error> {
    let worker = await!(query_broker());
    match worker {
        Some(worker) => await!(send_job(worker)),
        None => await!(process_job_locally(job))
    }
}
```

Design by community

- Open source
- RFC

  - RFC PR: rust-lang/rfcs#2394
  - Rust Issue: rust-lang/rust#50547

Summary

Add async & await syntaxes to make it more ergonomic to write code manipulating futures.

Backwards compatibility

- Strong BC guarantees

Backwards compatibility

- Strong BC guarantees
- New version every 6 weeks
  - Thousands of libraries tested to spot regressions

Backwards compatibility

- Strong BC guarantees
- New version every 6 weeks
  - Thousands of libraries tested to spot regressions
- Big changes => new edition
  - Rust 2015 vs Rust 2018

Unstable features

```
#![feature(async_await)]
async fn foo() {
    ...
}
```

Unstable features

```rust
#![feature(async_await)]
async fn foo() {
    ...
}
```

```
$ cargo +nightly build
```

Fast & Safe

Quick development

High performance

Zero-cost abstractions

Bjarne Stroustrup:

What you don’t use, you don’t pay for.
What you do use, you couldn’t hand code any better.

Minimal runtime

Minimal runtime

- No GC

Minimal runtime

- No GC

- No exceptions

Minimal runtime

- No GC

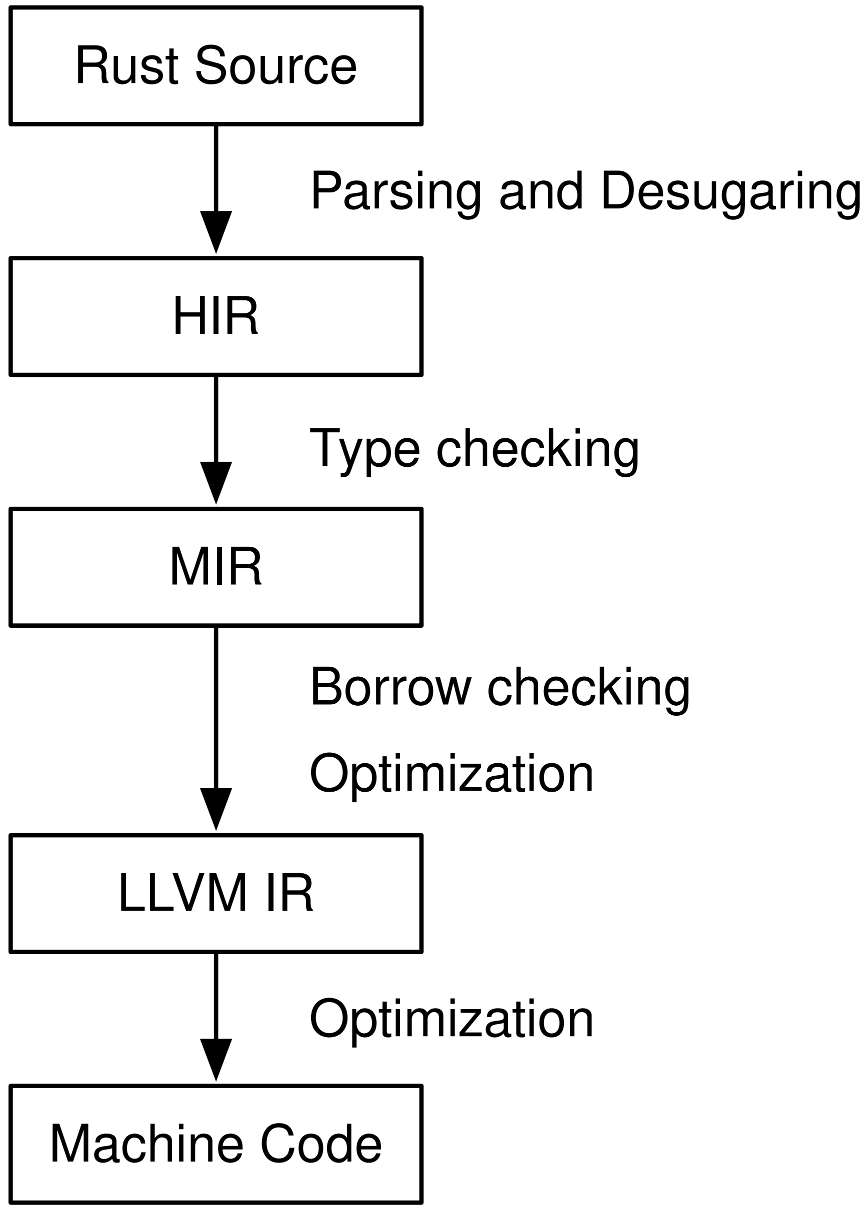
- No exceptions
- Tight data layout

Minimal runtime

- No GC

- No exceptions
- Tight data layout
- Supports embedded platforms

Compiles to LLVM



Compiles to LLVM

```
pub fn dot_product(x: &[f64], y: &[f64]) -> f64 {
    x.iter().zip(y).map(|(&a, &b)| a * b).sum::<f64>()
}
```

Compiles to LLVM

```rust
pub fn dot_product(x: &[f64], y: &[f64]) -> f64 {
    x.iter().zip(y).map(|(&a, &b)| a * b).sum::<f64>()
}
```

```asm
vmovsd   xmm1, qword ptr [rdi + 8*rsi]
vmovsd   xmm2, qword ptr [rdi + 8*rsi + 8]
vmulsd   xmm1, xmm1, qword ptr [rdx + 8*rsi]
vaddsd   xmm0, xmm0, xmm1
vmulsd   xmm1, xmm2, qword ptr [rdx + 8*rsi + 8]
vaddsd   xmm0, xmm0, xmm1
vmovsd   xmm1, qword ptr [rdi + 8*rsi + 16]
vmulsd   xmm1, xmm1, qword ptr [rdx + 8*rsi + 16]
vmovsd   xmm2, qword ptr [rdi + 8*rsi + 24]
vmulsd   xmm2, xmm2, qword ptr [rdx + 8*rsi + 24]
vaddsd   xmm0, xmm0, xmm1
vaddsd   xmm0, xmm0, xmm2
vmovsd   xmm1, qword ptr [rdi + 8*rsi + 32]
vmulsd   xmm1, xmm1, qword ptr [rdx + 8*rsi + 32]
vaddsd   xmm0, xmm0, xmm1
vmovsd   xmm1, qword ptr [rdi + 8*rsi + 40]
vmulsd   xmm1, xmm1, qword ptr [rdx + 8*rsi + 40]
vaddsd   xmm0, xmm0, xmm1
vmovsd   xmm1, qword ptr [rdi + 8*rsi + 48]
vmulsd   xmm1, xmm1, qword ptr [rdx + 8*rsi + 48]
vmovsd   xmm2, qword ptr [rdi + 8*rsi + 56]
```

Branch prediction

```
if core::intrinsic::likely(condition) {
    ...
} else #[cold] {
    ...
}
```

SIMD

```rust
#[cfg(target_arch = "x86_64")]
use std::arch::x86_64::_mm256_add_epi64;

_mm256_add_epi64(...);
```

SIMD

```rust
#[cfg(target_arch = "x86_64")]
use std::arch::x86_64::_mm256_add_epi64;

_mm256_add_epi64(...);
```

```rust
data.simd_iter()
    .simd_map(|v| {
        f32s(9.0) * v.abs().sqrt().ceil() -
        f32s(4.0) - f32s(2.0)
    })
    .scalar_collect();
```

Inline assembly

```
fn add(a: i32, b: i32) -> i32 {
    let c: i32;
    unsafe {
        asm!("add $2, $0"
             : "=r"(c)
             : "0"(a), "r"(b));
    }
    c
}
```

Constexpr functions

```
const fn double(x: i32) -> i32 {
    x * 2
}

const FIVE: i32 = 5;
const TEN: i32 = double(FIVE);
```

Concurrency primitives

- Mutexes

Concurrency primitives

- Mutexes
- Condition variables

Concurrency primitives

- Mutexes
- Condition variables
- Atomics

Concurrency primitives

- Mutexes
- Condition variables
- Atomics
- Synchronized queues

Shared-memory parallelism

No OpenMP ☹

Shared-memory parallelism

No OpenMP ☹

Rayon (+ Rayon adaptive )

```
fn sum_of_squares(input: &[i32]) -> i32 {
    input.par_iter()
         .map(|&i| i * i)
         .sum()
}
```

Shared-memory parallelism

No OpenMP ☹

Rayon (+ Rayon adaptive )

```
fn sum_of_squares(input: &[i32]) -> i32 {
    input.par_iter()
         .map(|&i| i * i)
         .sum()
}
```

Shared-memory parallelism

No OpenMP ☹

Rayon (+ Rayon adaptive )

```
fn sum_of_squares(input: &[i32]) -> i32 {
    input.par_iter()
         .map(|&i| i * i)
         .sum()
}

#[parallel]
for x in 0..10 {
    println!("{}", x);
}
```

Message-passing

```
let universe = mpi::initialize();
let world = universe.world();
let size = world.size();
let rank = world.rank();

if rank == 0 {
    let (msg, status) = world.any_process().receive_vec();
}
```

C/C++ interop

C from Rust

```
extern {
    fn snappy_max_compressed_length(len: size_t) -> size_t;
}

let length = unsafe { snappy_max_compressed_length(100) };
```

C/C++ interop

C from Rust

```
extern {
    fn snappy_max_compressed_length(len: size_t) -> size_t;
}

let length = unsafe { snappy_max_compressed_length(100) };
```

Rust from C

```
#[repr(C)]
struct Object {
    bar: i32,
}

extern "C" fn foo(param: *mut Object) {
    unsafe {
        (*target).bar = 5;
    }
}
```

Performance caveats

- Out-of-bounds checks

Performance caveats

- Out-of-bounds checks
  - Can be optimized away (iterators)

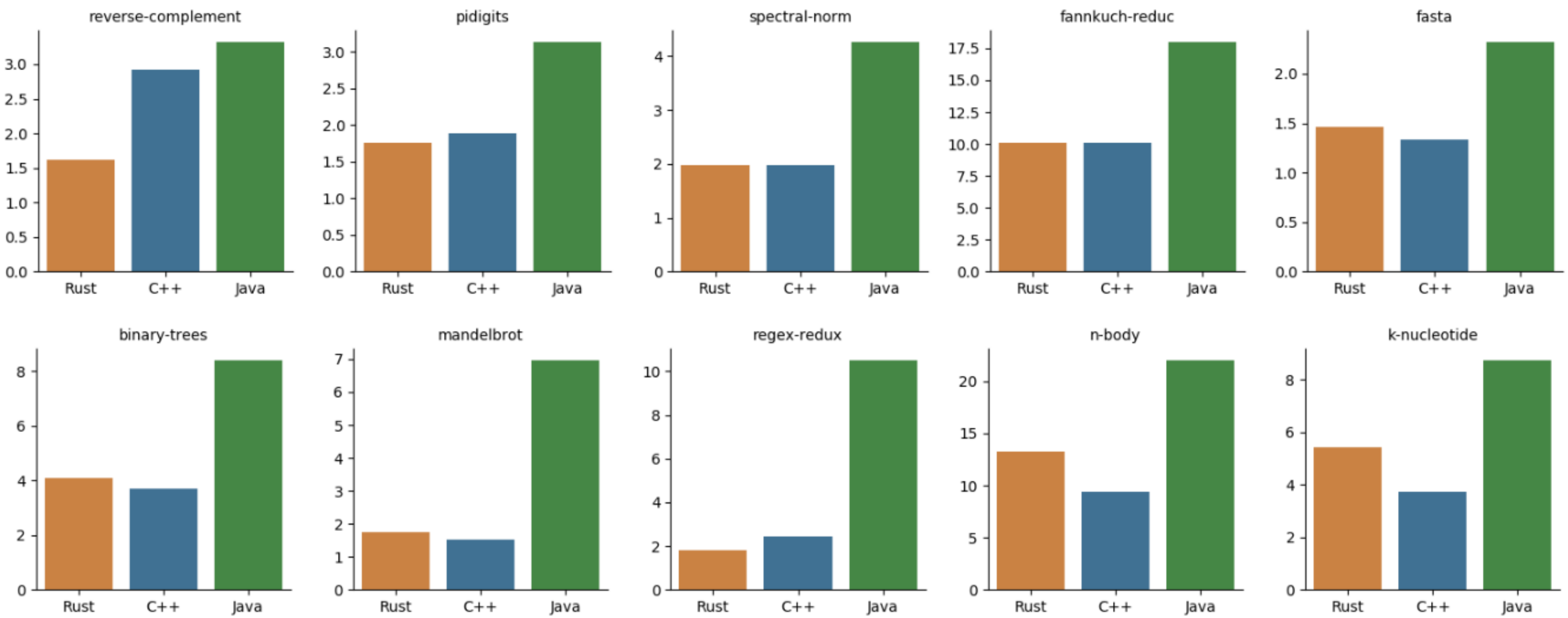
Performance caveats

- Out-of-bounds checks
  - Can be optimized away (iterators)
- Integer overflow is not undefined

Performance caveats

- Out-of-bounds checks
  - Can be optimized away (iterators)
- Integer overflow is not undefined
  - Runtime checks only in debug mode

Benchmark game



https://benchmarksgame-team.pages.debian.net/benchmarksgame/fastest/gpp-rust.html

Fast & Safe

Fast & Safe

Memory safety

Fast & Safe

Memory safety

Fearless concurrency

Fast & Safe

Memory safety

Fearless concurrency

Rust is safe…

Rust is safe…
…but from what?

Undefined behaviour

Working Draft, Standard for Programming Language C++

UB in Java

Java::Iterator::remove

"The behavior of an iterator is *unspecified* if the underlying collection is modified while the iteration is in progress in any way other than by calling this method, unless an overriding class has specified a concurrent modification policy."

UB in Python

`for` statement

"There is a subtlety when the sequence is being modified by the loop (this can only occur for mutable sequences, e.g. lists). An internal counter is used to keep track of which item … *This can lead to nasty bugs* that can be avoided by making a temporary copy using a slice of the whole sequence …"

Sources of UB

Sources of UB

- Null pointer dereference

Sources of UB

- Null pointer dereference
- Double-free

Sources of UB

- Null pointer dereference
- Double-free
- Use-after-free

Sources of UB

- Null pointer dereference
- Double-free
- Use-after-free
- Out-of-bounds access

Sources of UB

- Null pointer dereference
- Double-free
- Use-after-free
- Out-of-bounds access
- Integer conversion

Sources of UB

- Null pointer dereference
- Double-free
- Use-after-free
- Out-of-bounds access
- Integer conversion
- Integer overflow

Sources of UB

- Null pointer dereference
- Double-free
- Use-after-free
- Out-of-bounds access
- Integer conversion
- Integer overflow
- Iterator invalidation

Sources of UB

- Null pointer dereference
- Double-free
- Use-after-free
- Out-of-bounds access
- Integer conversion
- Integer overflow
- Iterator invalidation
- Invalid alignment

Sources of UB

- Null pointer dereference
- Double-free
- Use-after-free
- Out-of-bounds access
- Integer conversion
- Integer overflow
- Iterator invalidation
- Invalid alignment
- …

Sources of UB

- Null pointer dereference
- Double-free
- Use-after-free
- Out-of-bounds access
- Integer conversion
- Integer overflow
- Iterator invalidation
- Invalid alignment
- …

Rust tries very hard to avoid all of the above

Rust's insight

Rust's insight

Memory errors arise when aliasing is combined with mutability

C++ UB example

```cpp
std::vector<int> vec = { 1, 2, 3 };
```

C++ UB example

Aliasing ✓

```
std::vector<int> vec = { 1, 2, 3 };
int& p = vec[0];

std::cout << p << std::endl;
```

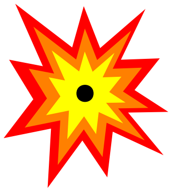
C++ UB example

Mutability ✓

```
std::vector<int> vec = { 1, 2, 3 };

vec.push_back(4);
```

C++ UB example

Aliasing & Mutability



```
std::vector<int> vec = { 1, 2, 3 };
int& p = vec[0];
vec.push_back(4);
std::cout << p << std::endl;
```

What to do?

Rust's solution

You can mutate

Rust's solution

You can mutate ||

Rust's solution

You can mutate || alias

Rust's solution

You can mutate || alias

But not both at the same time (w.r.t. a single variable)

Rust's solution

You can mutate || alias

But not both at the same time (w.r.t. a single variable)

Rust enforces this at compile time using its type system

Memory safety using the type system

- Ownership

Memory safety using the type system

- Ownership
- Borrowing

Memory safety using the type system

- Ownership
- Borrowing
- Lifetimes

Ownership

Every value in Rust has exactly one owner

Ownership

Every value in Rust has exactly one owner

When that owner goes out of scope, the value is dropped

Ownership

```
fn foo(bitmap: Bitmap) {
    ...
}
```

Ownership

No one else has any access to `bitmap`.
It can be mutated arbitrarily.

```
fn foo(bitmap: Bitmap) {
    ...
}
```

Ownership

```
fn foo(bitmap: Bitmap) {
    ...
} // bitmap is dropped here
```

Ownership - move semantics

```
fn foo(bitmap: Bitmap) { ... }

fn main() {
    let bitmap = Bitmap::load(...);
    foo(bitmap);
    ...
}
```

Ownership - move semantics

`bitmap` is moved here.
It will not be `dropped` in the current scope.

```
fn foo(bitmap: Bitmap) { ... }

fn main() {
    let bitmap = Bitmap::load(...);
    foo(bitmap);
    ...
}
```

Ownership - move semantics

```rust
fn foo(bitmap: Bitmap) { ... }

fn main() {
    let bitmap = Bitmap::load(...);
    foo(bitmap);
    println!("{}", bitmap.width);
}
```

Ownership - move semantics

```rust
fn foo(bitmap: Bitmap) { ... }

fn main() {
    let bitmap = Bitmap::load(...);
    foo(bitmap);
    println!("{}", bitmap.width);
}
```

```
error[E0382]: borrow of moved value: `bitmap`
  --> src/main.rs:11:20
10 |     foo(bitmap);
   |         ------ value moved here
11 |     println!("{}", bitmap.width);
   |                    ^^^^^^^^^^^^ value borrowed here after move
```

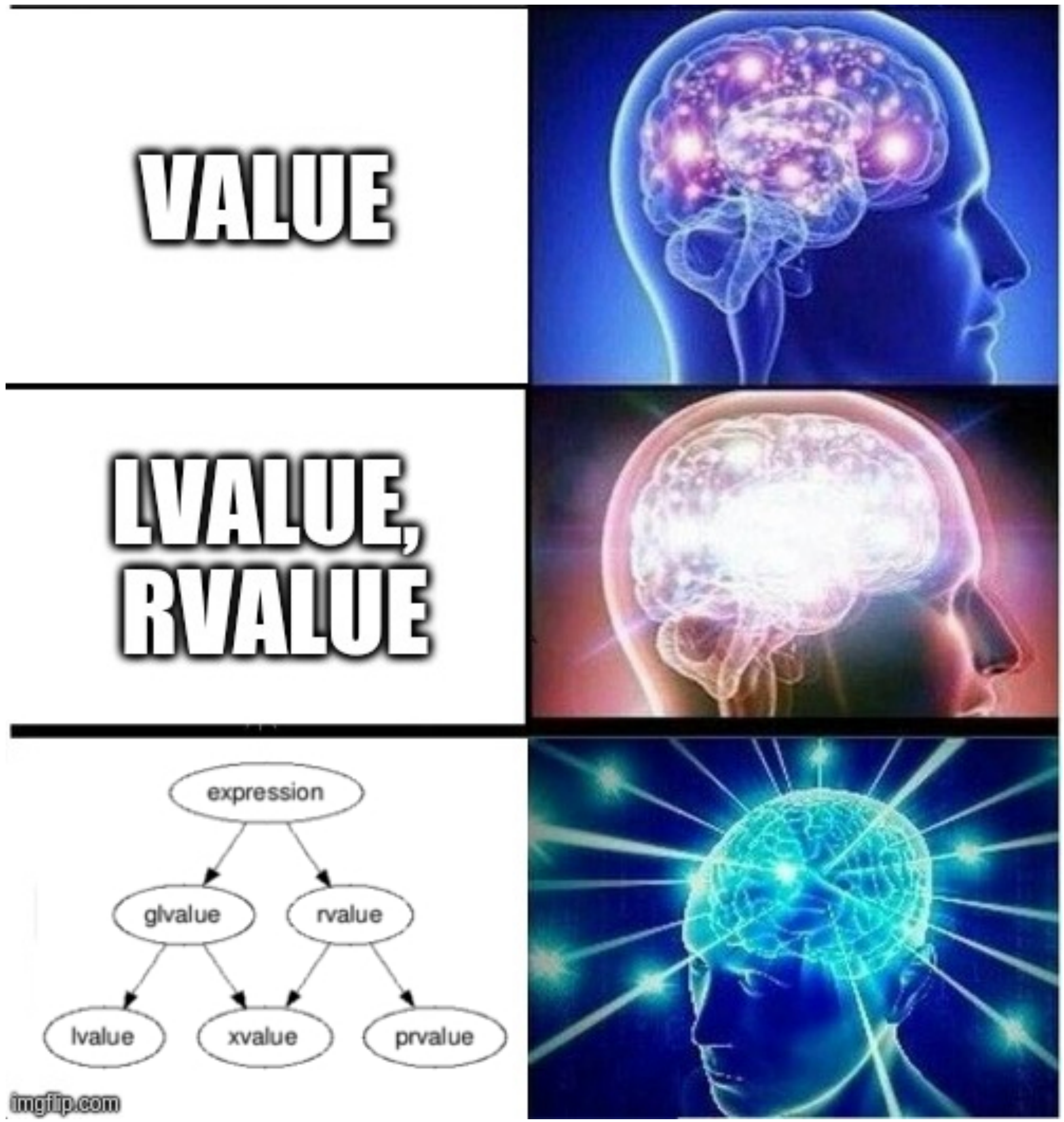
Constructors

- Move constructors? Nope.

Constructors

- Move constructors? Nope.
- Move assignment constructors? Nope.

Why are they needed in C++?



Why are they needed in C++?

```
void foo(Bitmap&& bitmap) { ... }

Bitmap bitmap(...);
foo(std::move(bitmap));
std::cout << bitmap.width << std::endl;
```

Why are they needed in C++?

```cpp
void foo(Bitmap&& bitmap) { ... }

Bitmap bitmap(...);
foo(std::move(bitmap));
std::cout << bitmap.width << std::endl;
```

`bitmap` is still accessible here.
It will be `dropped` at the end of scope.
Its state HAD to be reset in the move constructor.

"Copy" semantics

Values are copied instead of moved if they implement the `Copy` trait

"Copy" semantics

Values are copied instead of moved if they implement the `Copy` trait

Types are `Copy` if:

"Copy" semantics

Values are copied instead of moved
if they implement the `Copy` trait

Types are `Copy` if:

- they are primitive (integers, floats, etc.)

"Copy" semantics

**Values are copied instead of moved
if they implement the `Copy` trait**

Types are `Copy` if:

- they are primitive (integers, floats, etc.)
- they are marked as Copy

"Copy" semantics

Values are copied instead of moved if they implement the `Copy` trait

Types are `Copy` if:

- they are primitive (integers, floats, etc.)
- they are marked as Copy

```
#[derive(Copy)]
struct Person {
    age: u32,
    male: bool
}
```

"Copy" semantics

```rust
fn foo(num: u32) { ... }

let number = 5;
foo(number);
println!("{}", number); // no error
```

"Copy" semantics

```
fn foo(num: u32) { ... }

let number = 5;
foo(number);
println!("{}", number); // no error
```

`number` is copied here.
It can be still accessed after the call.

Where's the aliasing?

So far, we only have mutability, there's no aliasing:

Where's the aliasing?

So far, we only have mutability, there's no aliasing:

- After a move, the original value is not accessible

Where's the aliasing?

So far, we only have mutability, there's no aliasing:

- After a move, the original value is not accessible
- After a copy, a new value is created

Borrowing

Aliasing happens when you create a reference to a value

Borrowing

Aliasing happens when you create a reference to a value
This is called borrowing in Rust

Shared borrows

```
let value = Bitmap::load(...);
let a = &value;
let b = &value;
```

Shared borrows

```
let value = Bitmap::load(...);
let a = &value;
let b = &value;
```

- Multiple shared borrows of a value may exist

Shared borrows

```
let value = Bitmap::load(...);
let a = &value;
let b = &value;
```

- Multiple shared borrows of a value may exist
- You can't mutate using a shared borrow

```
a.width = 10; // does not compile
```

Shared borrows

```
let value = Bitmap::load(...);
let a = &value;
let b = &value;
```

- Multiple shared borrows of a value may exist
- You can't mutate using a shared borrow

```
a.width = 10; // does not compile
```

- You can't move out of a shared borrow

```
fn foo(bitmap: Bitmap) { }
foo(a); // does not compile
```

Unique borrows

```
let value = Bitmap::load(...);
let c = &mut value;
```

Unique borrows

```rust
let value = Bitmap::load(...);
let c = &mut value;
```

- If a unique borrow exists, there are no other references to the same value

Unique borrows

```
let value = Bitmap::load(...);
let c = &mut value;
```

- If a unique borrow exists, there are no other references to the same value
- You can only create a unique borrow if you own the value

Unique borrows

```
let value = Bitmap::load(...);
let c = &mut value;
```

- If a unique borrow exists, there are no other references to the same value
- You can only create a unique borrow if you own the value
- You can mutate using a unique borrow

```
c.width = 10;
```

Unique borrows

```
let value = Bitmap::load(...);
let c = &mut value;
```

- If a unique borrow exists, there are no other references to the same value
- You can only create a unique borrow if you own the value
- You can mutate using a unique borrow

```
c.width = 10;
```

- You can't move out of a unique borrow

```
fn foo(bitmap: Bitmap) { }
foo(c); // does not compile
```

Vector example (Rust)

```rust
let vec = vec!(1, 2, 3);
```

Vector example (Rust)

```rust
let vec = vec!(1, 2, 3);
let p = &vec[0];
```

Vector example (Rust)

```
// Vec::push
fn push(&mut self, value: T)
```

```
let vec = vec!(1, 2, 3);
let p = &vec[0];
vec.push(4);
```

Vector example (Rust)

```
// Vec::push
fn push(&mut self, value: T)
```

```
let vec = vec!(1, 2, 3);
let p = &vec[0];
vec.push(4);
println!("{}", p);
```

Vector example (Rust)

```
// Vec::push
fn push(&mut self, value: T)
```

```
let vec = vec!(1, 2, 3);
let p = &vec[0];
vec.push(4);
println!("{}", p);
```

```
error[E0502]: cannot borrow `vec` as mutable because it is also borrowed as immutable
  --> src/main.rs:11:5
   |
10 |     let p = &vec[0];
   |              --- immutable borrow occurs here
11 |     vec.push(4);
   |     ^^^^^^^^^^^ mutable borrow occurs here
12 |     println!("{}", *p);
   |                    -- immutable borrow later used here
   |
```

What if compile time is not enough?

If you can't prove to the compiler that your borrows are safe, borrow checking can be done at runtime.

What if compile time is not enough?

If you can't prove to the compiler that your borrows are safe,
borrow checking can be done at runtime.
If any rules are broken, the program panics.

What if compile time is not enough?

If you can't prove to the compiler that your borrows are safe, borrow checking can be done at runtime.
If any rules are broken, the program panics.

```
let value = RefCell::new(5);
let a = value.borrow();     // shared borrow
let b = value.borrow_mut(); // unique borrow
```

What if compile time is not enough?

If you can't prove to the compiler that your borrows are safe,
borrow checking can be done at runtime.
If any rules are broken, the program panics.

```
let value = RefCell::new(5);
let a = value.borrow();     // shared borrow
let b = value.borrow_mut(); // unique borrow
```

This would panic, since there already is
a shared borrow

Lifetimes (C++)

```cpp
int* p;
{
    int value = 5;
    p = &value;
}
std::cout << *p << std::endl;
```

Lifetimes (C++)

```cpp
int* p;
{
    int value = 5;
    p = &value;
} // <-- `value` is destroyed here
std::cout << *p << std::endl;
```

Lifetimes (Rust)

```rust
let p;
{
    let value = 5;
    p = &value;
}
println!("{}", *p);
```

Lifetimes (Rust)

```
let p;
{
    let value = 5;
    p = &value;
}
println!("{}", *p);
```

Lifetime of reference `p`

Lifetimes (Rust)

```rust
let p;
{
    let value = 5;
    p = &value;
}
println!("{}", *p);
```

Lifetime of reference `p`

Lifetime of `value`

Lifetimes (Rust)

```
let p;
{
    let value = 5;
    p = &value;
}
println!("{}", *p);
```

Lifetime of reference `p`

Lifetime of `value`

Lifetime of a value must be
>= lifetime of a reference to it

Lifetimes (Rust)

```rust
let p;
{
    let value = 5;
    p = &value;
}
println!("{}", *p);
```

```
error[E0597]: `value` does not live long enough
  --> src/main.rs:14:9
   |
14 |         p = &value;
   |         ^^^^^^^^^^ borrowed value does not live long enough
15 |     }
   |     - `value` dropped here while still borrowed
16 |     println!("{}", *p);
   |                    -- borrow later used here
```

What if compile time is not enough?

If you can't prove to the compiler that the lifetimes are correct, lifetime can be managed at runtime.

What if compile time is not enough?

If you can't prove to the compiler that the lifetimes are correct, lifetime can be managed at runtime.

```
fn main() {
    let value = Rc::new(5); // refcount == 1
```

What if compile time is not enough?

If you can't prove to the compiler that the lifetimes are correct, lifetime can be managed at runtime.

```
fn main() {
    let value = Rc::new(5); // refcount == 1
    {
        let a = value.clone(); // refcount == 2
```

What if compile time is not enough?

If you can't prove to the compiler that the lifetimes are correct, lifetime can be managed at runtime.

```
fn main() {
    let value = Rc::new(5); // refcount == 1
    {
        let a = value.clone(); // refcount == 2
    } // refcount == 1
```

What if compile time is not enough?

If you can't prove to the compiler that the lifetimes are correct, lifetime can be managed at runtime.

```
fn main() {
    let value = Rc::new(5); // refcount == 1
    {
        let a = value.clone(); // refcount == 2
    } // refcount == 1
} // refcount == 0, value is dropped
```

Fast & Safe

Memory safety

Fearless concurrency

Concurrency issues

Rust doesn't prevent:

Concurrency issues

Rust doesn't prevent:

- Deadlocks

Concurrency issues

Rust doesn't prevent:

- Deadlocks
- General race conditions

Concurrency issues

Rust doesn't prevent:

- Deadlocks
- General race conditions

Rust prevents (at compile time):

Concurrency issues

Rust doesn't prevent:

- Deadlocks
- General race conditions

Rust prevents (at compile time):

- Data races

What causes data races?

What causes data races?

Concurrent aliasing and mutability...

What causes data races?

Concurrent aliasing and mutability...
...but Rust already disables that!

What causes data races?

Concurrent aliasing and mutability…
…but Rust already disables that!

So how do we get any concurrency at all…?

Spawning a thread

```
fn spawn<F: Fn + Send>(f: F)
```

Spawning a thread

```
fn spawn<F: Fn + Send>(f: F)
```

Ownership of T can be transferred to another thread only if T implements the *Send* trait

Spawning a thread

```
fn spawn<F: Fn + Send>(f: F)
```

Ownership of T can be transferred to another thread
only if T implements the *Send* trait

Send is implemented automatically, unless the type
contains values that are not safe to be transferred between threads

Shared state concurrency

Goal:

Shared state concurrency

Goal:

- Spawn a thread

Shared state concurrency

Goal:

- Spawn a thread
- Send a reference to some value to it

Shared state concurrency

Goal:

- Spawn a thread
- Send a reference to some value to it
- Modify the value in the spawned thread

Shared state concurrency

Goal:

- Spawn a thread
- Send a reference to some value to it
- Modify the value in the spawned thread
- Read the value in the original thread

Shared state concurrency

```
let value = 5;
```

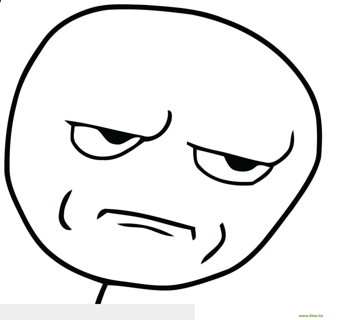
Shared state concurrency

```
let value = 5;
let p = &value;
```

Shared state concurrency

```
let value = 5;
let p = &value;
thread::spawn(|| {
    println!("{}", *p);
});
```

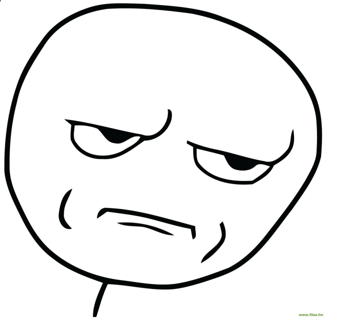
Shared state concurrency



```
let value = 5;
let p = &value;
thread::spawn(|| {
    println!("{}", *p);
});
```

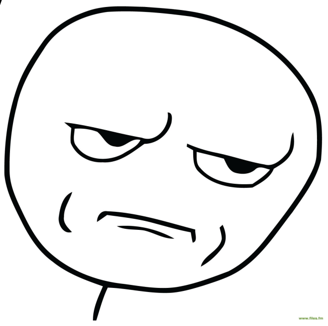
```
error[E0373]: closure may outlive the current function,
 but it borrows `p`, which is owned by the current function

  --> src/main.rs:17:19
   |
17 |     thread::spawn(|| {
   |                   ^^ may outlive borrowed value `p`
18 |         println!("{}", *p);
   |                         - `p` is borrowed here
```

Shared state concurrency



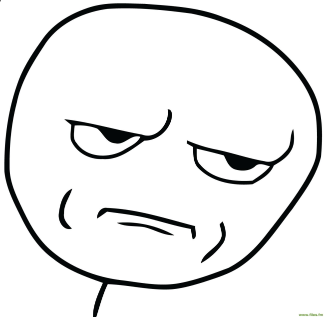
```
let p = Rc::new(5);
```

Shared state concurrency



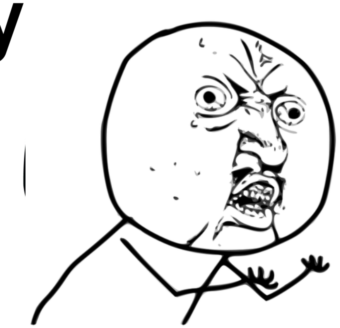
```
let p = Rc::new(5);
thread::spawn(|| {
```

Shared state concurrency



```
let p = Rc::new(5);
thread::spawn(|| {
    println!("{}", *p);
});
```

Shared state concurrency



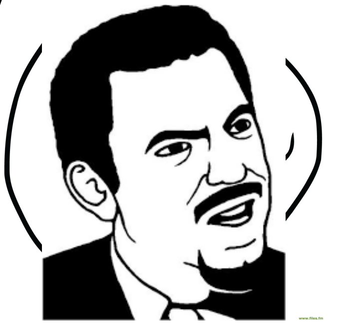
```rust
let p = Rc::new(5);
thread::spawn(|| {
    println!("{}", *p);
});
```

```
error[E0373]: closure may outlive the current function,
  --> src/main.rs:16:19
   |
16 |     thread::spawn(|| {
   |                   ^^ may outlive borrowed value `p`
17 |         println!("{}", *p);
   |                         - `p` is borrowed here
```

Shared state concurrency

```
let p = Rc::new(5);
thread::spawn(move || {
    println!("{}", *p);
});
```

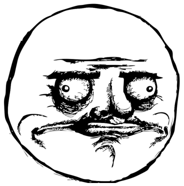
Shared state concurrency



```
let p = Rc::new(5);
thread::spawn(move || {
    println!("{}", *p);
});
```

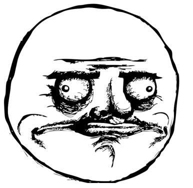
```
error[E0277]: `std::rc::Rc<i32>` cannot be shared between threads safely
  --> src/main.rs:16:5
   |
16 |     thread::spawn(|| {
   |     ^^^^^^^^^^^^^ `std::rc::Rc<i32>` cannot be shared between threads safely
   |
   = help: the trait `std::marker::Sync` is not implemented for `std::rc::Rc<i32>`
```

Shared state concurrency



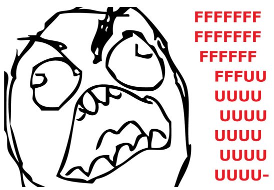
```
let p = Arc::new(5);
thread::spawn(move || {
    println!("{}", *p);
});
```

Shared state concurrency



```
let p = Arc::new(5);
thread::spawn(move || {
    println!("{}", *p);
});
println!("{}", *p);
```

Shared state concurrency




```rust
let p = Arc::new(5);
thread::spawn(move || {
    println!("{}", *p);
});
println!("{}", *p);
```

```
error[E0382]: borrow of moved value: `p`
  --> src/main.rs:19:21
   |
15 |     let p = Arc::new(5);
   |         - move occurs because `p` has type `std::sync::Arc<i32>`, which does not implement the `Copy` trait
16 |     thread::spawn(move || {
   |                   ------- value moved into closure here
17 |         println!("{}", *p);
   |                         - variable moved due to use in closure
18 |     });
19 |     println!("{}", *p);
   |                     ^ value borrowed here after move
```

Shared state concurrency

```
let p = Arc::new(5);
let tp = p.clone();
thread::spawn(move || {
    println!("{}", *tp);
});
println!("{}", *p);
```

Shared state concurrency

```
fn clone(&self) -> Arc<T>;
```

```
let p = Arc::new(5);
let tp = p.clone();
thread::spawn(move || {
    println!("{}", *tp);
});
println!("{}", *p);
```

Clone() creates a new Arc.
Multiple variables remove aliasing.

Shared state concurrency

```
fn clone(&self) -> Arc<T>;
```

```
let p = Arc::new(5);
let tp = p.clone();
thread::spawn(move || {
    println!("{}", *tp);
});
println!("{}", *p);
```

Clone() creates a new Arc.
Multiple variables remove aliasing.
Arc only provides *read-only* access (shared borrow).

Shared state concurrency

```
Mutex::new(5)
```

Shared state concurrency

```
let p = Arc::new(Mutex::new(5));
```

Shared state concurrency

```rust
let p = Arc::new(Mutex::new(5));
let tp = p.clone();
thread::spawn(move || {
    *tp.lock() = 10;
});
println!("{}", *p.lock());
```

Shared state concurrency

```
// Mutex::lock
fn lock(&self) -> &mut T;
```

```
let p = Arc::new(Mutex::new(5));
let tp = p.clone();
thread::spawn(move || {
    *tp.lock() = 10;
});
println!("{}", *p.lock());
```

Message passing

```
            mpsc::channel();
```

Message passing

```rust
let (tx, rx) = mpsc::channel();
```

Message passing

```
let (tx, rx) = mpsc::channel();
thread::spawn(move || {
    tx.send(5);
});
```

Message passing

```rust
let (tx, rx) = mpsc::channel();
thread::spawn(move || {
    tx.send(5);
});
let received = rx.recv();
```

Message passing

```
let (tx, rx) = mpsc::channel();
thread::spawn(move || {
    tx.send(5);
});
let received = rx.recv();
```

Splitting a channel into a receiver + sender removes aliasing and allows moving the sender independently of the receiver.

Where's the catch?

We have seen things that mutate through a shared borrow

Where's the catch?

We have seen things that mutate through a shared borrow

```
// Arc::clone
fn clone(&self) -> Arc<T>;
```

Where's the catch?

We have seen things that mutate through a shared borrow

```
// Arc::clone
fn clone(&self) -> Arc<T>;
// Mutex::lock
fn lock(&self) -> &mut T;
```

Where's the catch?

We have seen things that mutate through a shared borrow

```
// Arc::clone
fn clone(&self) -> Arc<T>;
// Mutex::lock
fn lock(&self) -> &mut T;
// AtomicU64::store
fn store(&self, val: u64, order: Ordering);
```

Where's the catch?

We have seen things that mutate through a shared borrow

```rust
// Arc::clone
fn clone(&self) -> Arc<T>;
// Mutex::lock
fn lock(&self) -> &mut T;
// AtomicU64::store
fn store(&self, val: u64, order: Ordering);
```

This is called `interior mutability` and requires unsafe Rust

Enter unsafe Rust

Some scenarios are not expressible in (safe) Rust

Enter unsafe Rust

Some scenarios are not expressible in (safe) Rust

In some cases, something more is required to:

Enter unsafe Rust

Some scenarios are not expressible in (safe) Rust

In some cases, something more is required to:

- Express inherently unsafe paradigms

Enter unsafe Rust

Some scenarios are not expressible in (safe) Rust

In some cases, something more is required to:

- Express inherently unsafe paradigms
- Improve performance

Enter unsafe Rust

Some scenarios are not expressible in (safe) Rust

In some cases, something more is required to:

- Express inherently unsafe paradigms
- Improve performance
- Interact with I/O, OS, hardware, network

Unsafe Rust

You can mark parts of code with the `unsafe` keyword

Unsafe Rust

You can mark parts of code with the `unsafe` keyword

Unsafe Rust is a *superset* of Rust

Unsafe Rust

Unsafe Rust allows:

Unsafe Rust

Unsafe Rust allows:

Accessing a global mutable variable

```
static mut COUNTER: u32 = 0;

fn increment_count() {
    unsafe {
        COUNTER += 1;
    }
}
```

Unsafe Rust

Unsafe Rust allows:

Dereferencing a raw pointer

```
let ptr = 0xCAFECAFE as *mut u32;
unsafe {
    *ptr = 5;
}
```

Unsafe Rust

Unsafe Rust allows:

Calling an unsafe function

```
unsafe {
    zlib_compress(&buffer, buffer.len());
}
```

Unsafe Rust

Unsafe Rust allows:

Implementing an unsafe trait

```
unsafe impl Send for MySuperSafeType {
    ...
}
```

Finding unsafe code - C++

```
std::atomic<LifecycleId> ArenaImpl::lifecycle_id_generator_;
GOOGLE_THREAD_LOCAL ArenaImpl::ThreadCache ArenaImpl::thread_cache_ = {-1, NULL};

void ArenaImpl::Init() {
  lifecycle_id_ =
      lifecycle_id_generator_.fetch_add(1, std::memory_order_relaxed);
  hint_.store(nullptr, std::memory_order_relaxed);
  threads_.store(nullptr, std::memory_order_relaxed);

  if (initial_block_) {
    // Thread which calls Init() owns the first block. This allows the
    // single-threaded case to allocate on the first block without having to
    // perform atomic operations.
    new (initial_block_) Block(options_.initial_block_size, NULL);
    SerialArena* serial =
        SerialArena::New(initial_block_, &thread_cache(), this);
    serial->set_next(NULL);
    threads_.store(serial, std::memory_order_relaxed);
    space_allocated_.store(options_.initial_block_size,
                           std::memory_order_relaxed);
    CacheSerialArena(serial);
  } else {
    space_allocated_.store(0, std::memory_order_relaxed);
  }
}
```

Finding unsafe code - C++

```
std::atomic<LifecycleId> ArenaImpl::lifecycle_id_generator_;
GOOGLE_THREAD_LOCAL ArenaImpl::ThreadCache ArenaImpl::thread_cache_ = {-1, NULL};

void ArenaImpl::Init() {
  lifecycle_id_ =
      lifecycle_id_generator_.fetch_add(1, std::memory_order_relaxed);
  hint_.store(nullptr, std::memory_order_relaxed);
  threads_.store(nullptr, std::memory_order_relaxed);

  if (initial_block_) {
    // Thread which calls Init() owns the first block. This allows the
    // single-threaded case to allocate on the first block without having to
    // perform atomic operations.
    new (initial_block_) Block(options_.initial_block_size, NULL);
    SerialArena* serial =
        SerialArena::New(initial_block_, &thread_cache(), this);
    serial->set_next(NULL);
    threads_.store(serial, std::memory_order_relaxed);
    space_allocated_.store(options_.initial_block_size,
                           std::memory_order_relaxed);
    CacheSerialArena(serial);
  } else {
    space_allocated_.store(0, std::memory_order_relaxed);
  }
}
```

Finding unsafe code - Rust

```
$ grep "unsafe" main.rs
```

Rust builds safe abstractions on top of unsafe foundations

Thanks, our curse has finally been lifted

Thanks, our curse has finally been lifted

Now YOU have to go and spread the word about Rust