Writing Safer, Faster, And Highly Concurrent Code With Rust

Jared M. Smith
@jaredthecoder

KCDC 2017

Thanks to the sponsors of KCDC!

TITANIUM SPONSORS

epiQ

PAIGE TECHNOLOGIES
INTELLIGENT PAIRING. PERPETUAL SUCCESS.

ADAPTIVE SOLUTIONS GROUP

okta

Platinum Sponsors

VinSolutions
Make every connection count.

jack henry & ASSOCIATES INC.

VALOREM

NAIC
National Association of Insurance Commissioners

NIPR
NATIONAL INSURANCE PRODUCER REGISTRY

TEKsystems
Our people make IT possible.

KEYHOLE SOFTWARE

smartsheet
Work Better™

Couchbase

EquipmentShare

KOCH™

PLURALSIGHT

Google Cloud

VML

mylo™
POWERED BY LOCKTON

Gold Sponsors

Advantage Tech
IT Staffing & Recruiting Services

PeopleAdmin

CENTRIQ TRAINING

Cerner

KU EDWARDS CAMPUS
The University of Kansas

twilio

{Track:js}
JAVASCRIPT ERROR MONITORING

OBJECT PARTNERS

smg service management group®

stackify

GARMIN

Balance Innovations

Allied Global SERVICES

dsi

TouchNet + Heartland

Mediware

LRS
CONSULTING SERVICES

Aviture
ingenuity. engineered.

An affiliate of C&A Industries, Inc.
AUREUS GROUP
Finance | Systems | Executive

Children's Mercy KANSAS CITY

Commerce Bank

SECURITY PS
STRATEGIC INFORMATION SECURITY

ProKarma

Help me help you!

There are note cards on each row and in the back. When this is over, leave one on the back table with any comments.

GREEN - Great job!
YELLOW - It was okay, here's why.
PINK - It was terrible, here's why.

About Me

- From Knoxville, TN
- Cyber Security Researcher and Project Lead at Oak Ridge National Laboratory
- CS PhD Student at the University of Tennessee, Knoxville
- Guest Author at Treehouse
- Avid hiker and reluctant backpacker

OAK RIDGE National Laboratory

University of Tennessee VOLSEC — COMPUTER SECURITY LAB —

treehouse™

my happy place

Agenda

- Background
- Why Rust?
- Rust Features
- Rust in the Wild
- Let’s write a web API!
- Q&A

Background

Where did we come from;
where did we go?

A long time ago in a computer research lab far, far away, the C programming language was invented by Dennis Ritchie...

1970

At Bell Laboratories, where many other amazing things were made, like UNIX, the semiconductor, and telephone.

We use C/C++ everywhere!

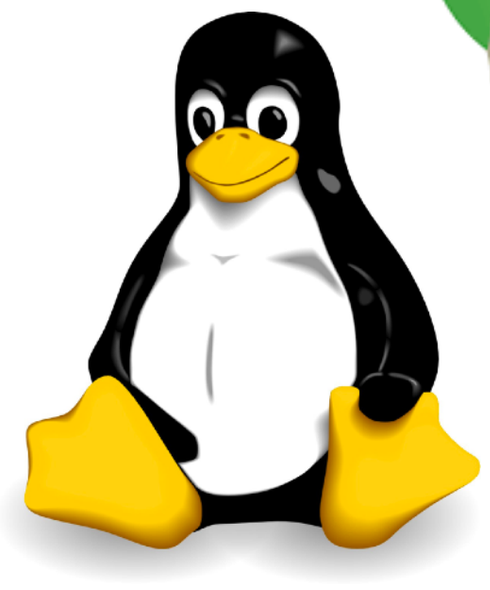
Where we need great performance, tight control over memory, or highly concurrent or parallel applications

OpenSSL™
Cryptography and SSL/TLS Toolkit

PostgreSQL

MySQL®

mongoDB.



JS

python™

Windows

php

@jaredthecoder | KCDC 2017

C/C++ gives us great things:

- Direct Memory Access
- Zero Cost Abstractions
- No Garbage Collection
- Compiled

However...

DIRTY COW

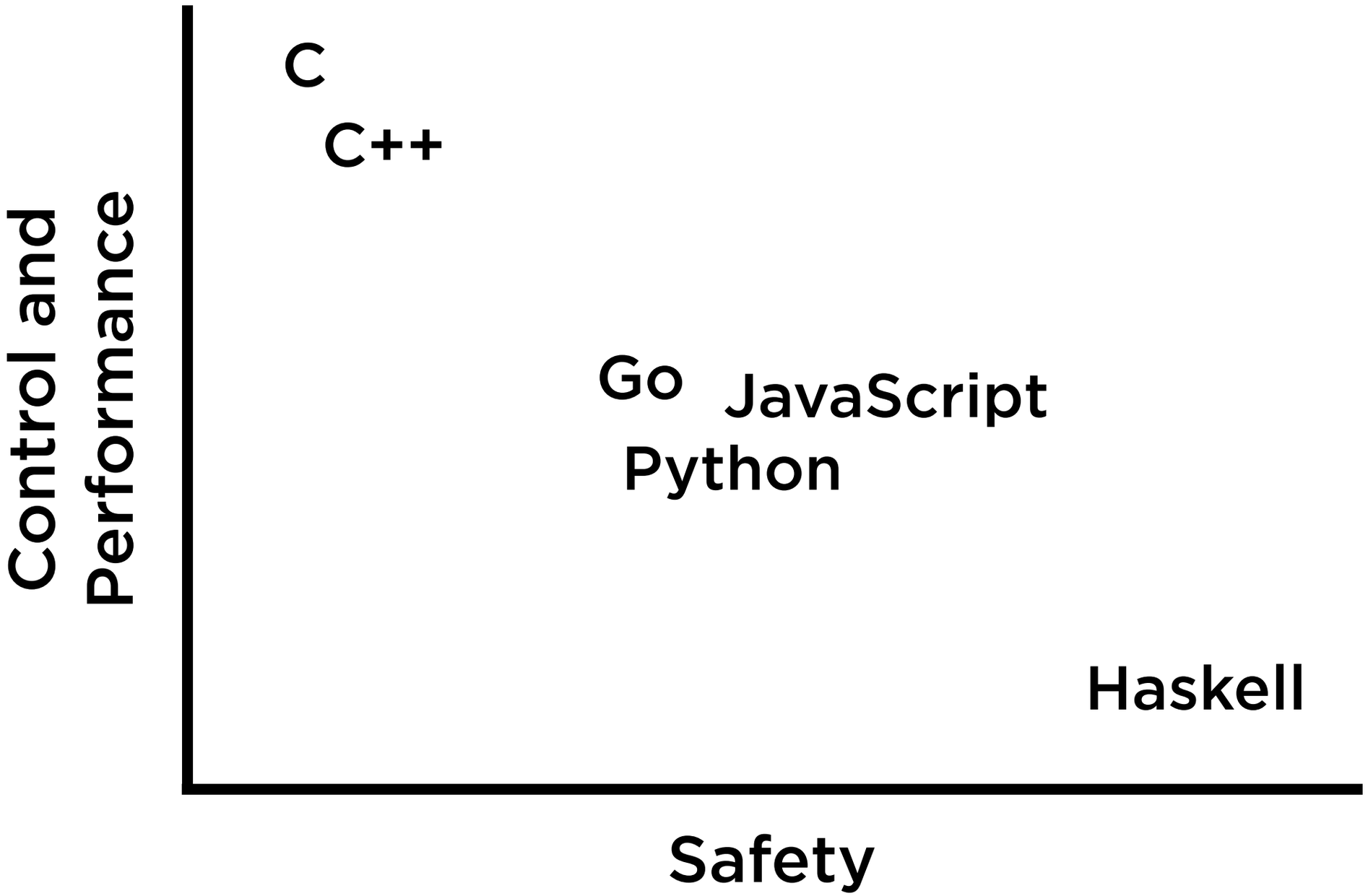
@jaredthecoder | KCDC 2017

@jaredthecoder | KCDC 2017

There are many bad things:

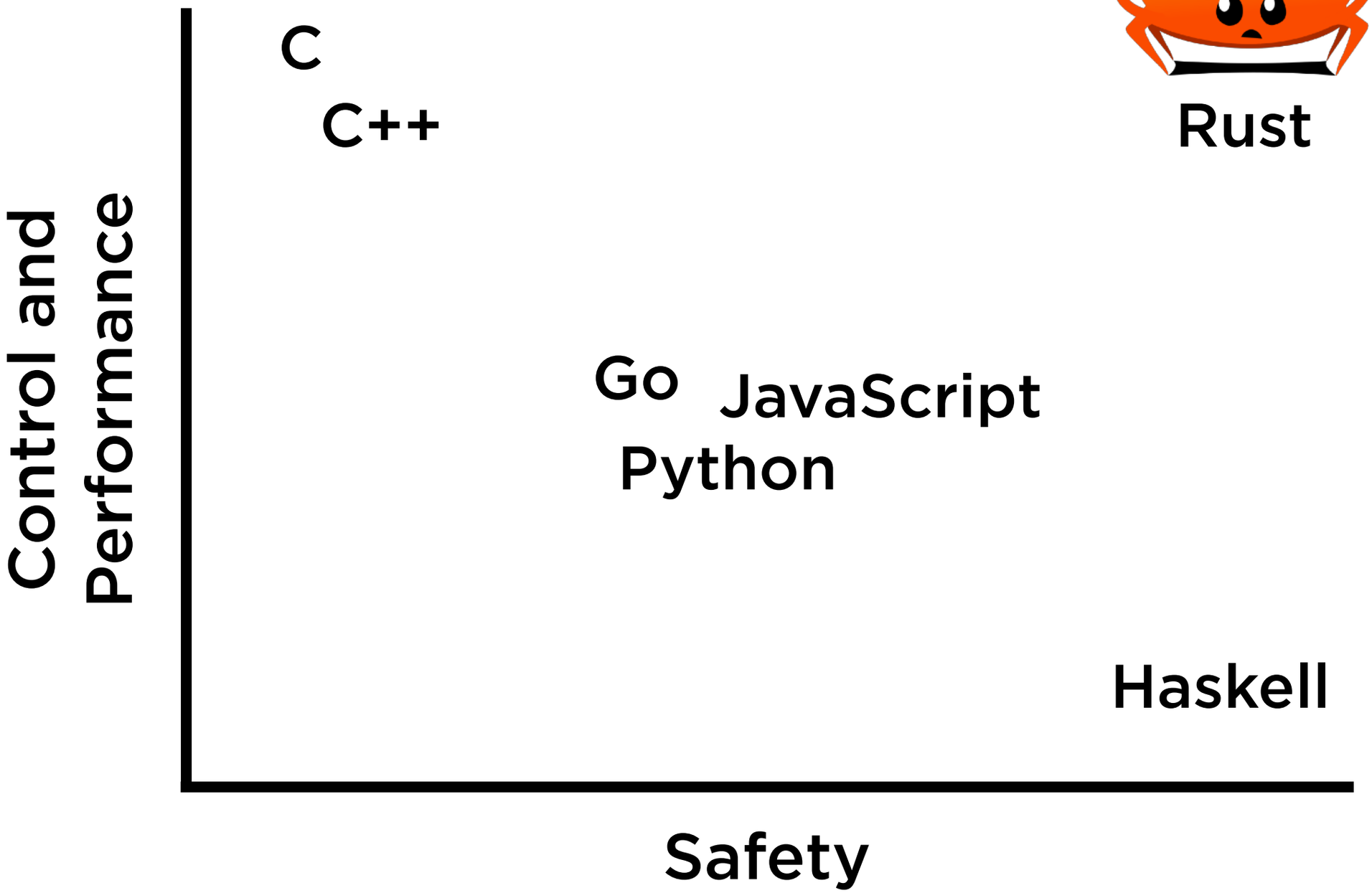
- Manual memory management
- Segfaults
- Data Races
- Tedious to write

Languages like
Python, Java, and
JavaScript save us
from much of this...


Control and
Performance
C
C++
Go
JavaScript
Python
Haskell
Safety

“Rust is a systems programming language that runs blazingly fast, prevents segfaults, and guarantees thread safety.”

- https://www.rust-lang.org


Control and Performance
C
C++
Rust
Go
JavaScript
Python
Haskell
Safety

Rust

Why though?

What's Rust?

- Safe systems language "disguised" as a high-level language
- Rich typesystem
- Statically-typed with type inference
- Strong functional programming influence
- Extensive tooling and well-maintained 3rd-party ecosystem

Rust's Governance Model

- Started as a personal project at Mozilla
- Owned by the community and backed by Mozilla
- Lots of external contributors
- Open RFC process

Complexity versus Speed

*Rust is now approaching or faster than C for many realistic applications

Easier

PHP

Python

Go/Java

Rust

C*

Faster

Key Language Features

- **Immutability**
  - All data is immutable by default
- **Ownership***
  - Every piece of data has an owner
- **Borrowing**
  - To share data, it's ownership must be transferred
- **Lifetimes**
  - Every piece of data can only be used for so long and can only be borrowed in certain ways

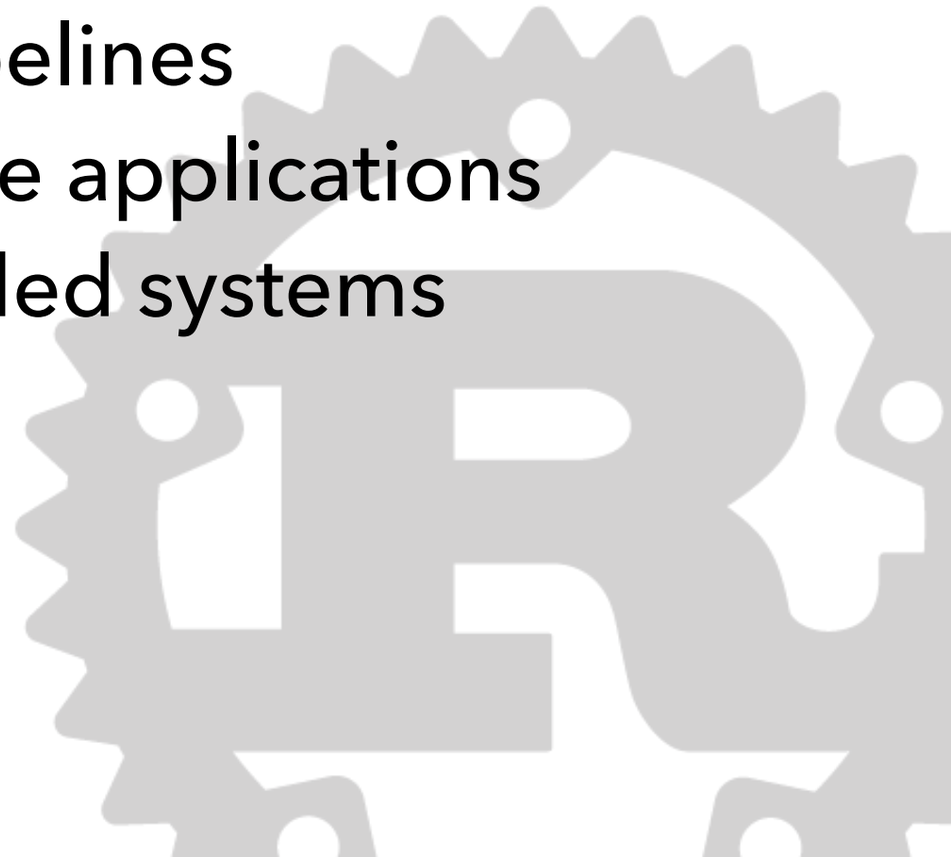
Key Language Features

- **Pattern Matching**
  - Eliminate sequences of complex conditional statements
- **Traits and Generics**
  - Allow for clean code by allowing shared behavior
- **Option and Result Types**
  - No null pointer exceptions!

Where Rust Excels

- Anywhere you want speed and safety, with the ability to write code quickly

  - Servers
  - Operating Systems
  - Compilers
  - Emulators
  - Finance
  - Cryptography
  - Web apps
  - Databases
  - Graphics Programming
  - Data Pipelines
  - Real-time applications
  - Embedded systems



Rust's Features

What's in it for me?

Basics

Expression-oriented language

Data is
immutable by
default!

Statically-typed, Inferred

@jaredthecoder | KCDC 2017

Algebric types

-> Enums!

Control Flow

- Basic Control Flow
  - *if*, *else*
  - *loop*, *while*, *for*
- Functions
  - *fn*
  - closures

Match Statements

```
let number = 4;
match number {
    1 => println!("One!"),
    2 | 3 | 5 | 7 | 11 => println!("This is a prime"),
    13...19 => println!("A teen"),
    _ => println!("Ain't special"),
}
```

Match Statements

```
let boolean = true;
let binary = match boolean {
    false => 0,
    // true => 1,
};
```

Match Statements

```
 › cargo run
   Compiling kcdc v0.1.0 (file:///Users/jared/development/rust/projects/kcdc)
error[E0004]: non-exhaustive patterns: `true` not covered
 --> src/main.rs:6:24
  |
6 |     let binary = match boolean {
  |                        ^^^^^^^ pattern `true` not covered
```

Containers and “Inheritance”

Structs

- Structs are lightweight classes

```
struct State;
```

Containers

- Add methods with *impl*

```
impl State {
    fn secede(&self) {
        print!("We, {}, secede from the union!", self.name);
    }
}
```

Structs

- They can have fields

```
struct State {
    name: String,
    capital: String,
    population: i64,
    income_tax: bool
}
```

Structs

- They can be instantiated

```
let kansas = State {
    name: "Kansas".to_owned(),
    capital: "Topeka".to_owned(),
    population: 2_907_000,
};
```

Structs

- But not partially instantiated!

```
› cargo run
   Compiling kcdc v0.1.0 (file:///Users/jared/development/rust/projects/kcdc)
error[E0063]: missing field `income_tax` in initializer of `State`
 --> src/main.rs:2:18
  |
2 |     let kansas = State {
  |                  ^^^^^ missing `income_tax`
```

Structs

- Rust has the *Option* Enum for this

```
struct State {
    name: String,
    capital: String,
    population: i64,
    income_tax: Option<bool>
}
```

Structs

- No more NullPointerExceptions!

```
let kansas = State {
    name: "Kansas".to_owned(),
    capital: "Topeka".to_owned(),
    population: 2_907_000,
    income_tax: None
};
```

- When you unpack the *income_tax*, parameter, you are forced to handle the case where it is None.

Traits

- Let's define two other structs

```
struct County {
    name: String,
    seat: Option<String>,
    population: Option<i64>
}

struct Country {
    name: String,
    capital: Option<String>,
    population: Option<i64>
}
```

Traits

- Traits are similar to interfaces, they define a contract for a type to abide by

```
trait Elect {
    fn elect(&self, leader: String);
}
```

Traits

- We can implement traits for our structs

```
impl Elect for State {
    fn elect(&self, leader: String) {
        println!("We elect {} for Governor!", leader);
    }
}

impl Elect for County {
    fn elect(&self, leader: String) {
        println!("We elect {} for Mayor!", leader);
    }
}
```

Traits

- Then we can use it's interface in the instance

```
let kansas = State { name: "Kansas".to_owned() };
let wyandotte = County { name: "Wyandotte".to_owned() };
let usa = Country { name: "United States of America".to_owned() };

kansas.elect("Michael Scott".to_owned());
wyandotte.elect("Kelly Kapoor".to_owned());
usa.elect("Dwight Schrute".to_owned());
```

Traits

- Unless we didn't implement it!

```
» cargo run                                                    [13:27:49]
   Compiling kcdc v0.1.0 (file:///Users/jared/development/rust/projects/kcdc)
error: no method named `elect` found for type `Country` in the current scope
 --> src/main.rs:9:9
  |
9 |     usa.elect("Dwight Schrute".to_owned());
  |         ^^^^^
  |
  = help: items from traits can only be used if the trait is implemented and in scope; the
following trait defines an item `elect`, perhaps you need to implement it:
  = help: candidate #1: `Elect`
```

Traits

- Default implementations are possible

```
trait Elect {
    fn elect(&self, leader: String) {
        println!("I, for one, welcome our new robot overlords.");
    }
}

impl Elect for Country {}
```

Traits

- You can define traits on foreign types

```
use std::ops::{Range, RangeFrom, RangeTo};

pub trait RangeExt<T> {
    fn before(&self) -> RangeTo<T>;
    fn after(&self) -> RangeFrom<T>;
}

impl<T: Copy> RangeExt<T> for Range<T> {
    fn before(&self) -> RangeTo<T> {
        ..self.start
    }

    fn after(&self) -> RangeFrom<T> {
        self.end..
    }
}
```

Traits

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

    #[test]
    fn range_ext() {
        assert_eq!(..3, (3..5).before());
        assert_eq!(5.., (3..5).after());
    }
}
```

Traits

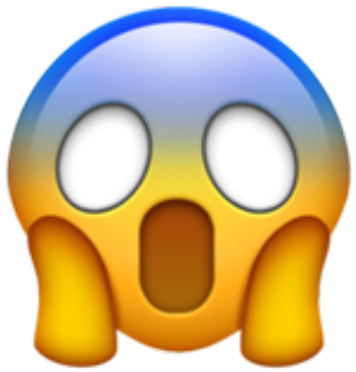
```
› cargo test                                                    [11:12:57]
   Finished dev [unoptimized + debuginfo] target(s) in 0.0 secs
    Running target/debug/deps/kcdc-bc70f91b4a266db1

running 1 test
test tests::range_ext ... ok

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

Ownership, Borrowing, and Lifetimes

@jaredthecoder | KCDC 2017



Garbage Collection

- Garbage collectors clean up memory that is no longer used
- Eliminates unreachable objects from memory by tracing root objects
- At the core of C# (CLR), Java (JVM), Python, Go, JavaScript (V8), Ruby, etc.

Garbage Collection Drawbacks

- Must run background threads for the GC
- May require large memory space

Reference Counting

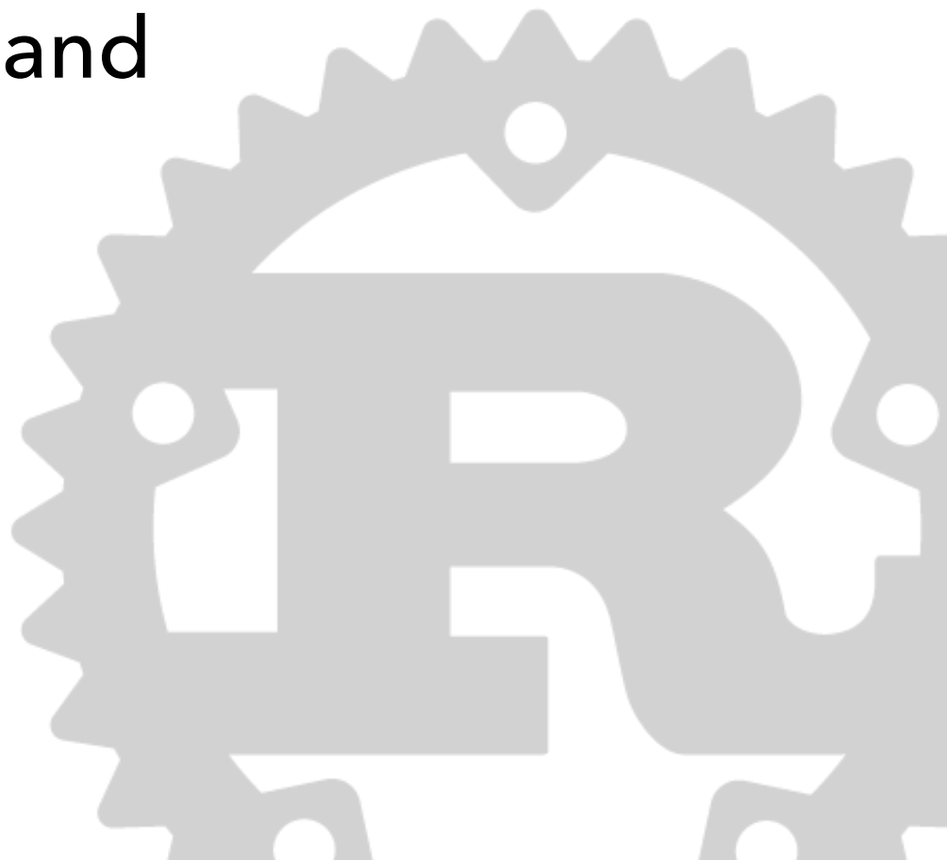
- Automated Reference Counting (ARC) keeps count on references of objects
- Deallocates them when they're no longer referenced
- Swift is a notable example

ARC Drawbacks

- Every assignment/allocation of variable:
  - needs to increment/decrement counters
  - reassign weak references
  - call destructors

Memory Safety

- Garbage collectors, ARC, and other similar memory cleanup methods keep our programs safe
- No segfaults, use-after-free, etc. in those languages
- However, they incur cost at runtime and data races can still exist



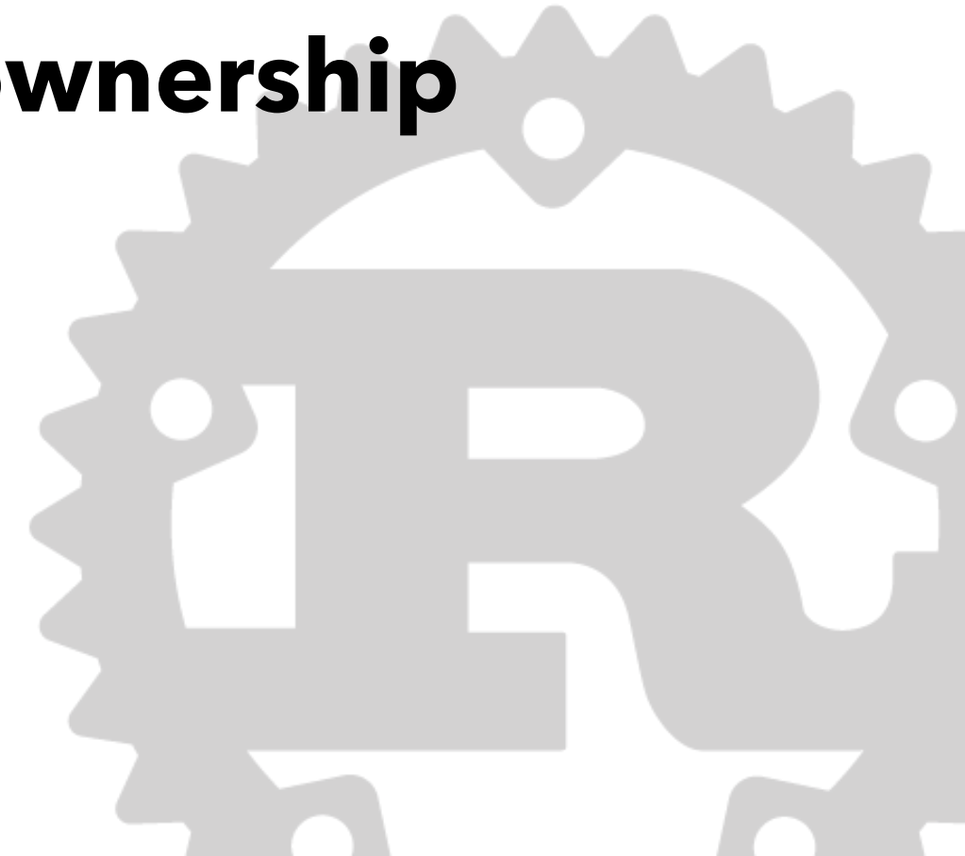
Still safe

ZERO runtime cost

@jaredthecoder | KCDC 2017

Rust to the Rescue!

- Rust provides the same and often stronger protection than garbage collection/ARC
- With **zero performance cost** at runtime
  - Instead, enforces rules to protect memory at **compile-time**
- It does this with it's system of **ownership**



ENTERING DANGER ZONE

Ownership: You own a resource, and when you are done with it, that resource is no longer in scope and gets deallocated.

References to a resource depend on the **lifetime** of that resource.

(they are only valid until the resource is deallocated)

Move semantics: Giving an owned resource to a function means giving it away, and you can no longer access it.

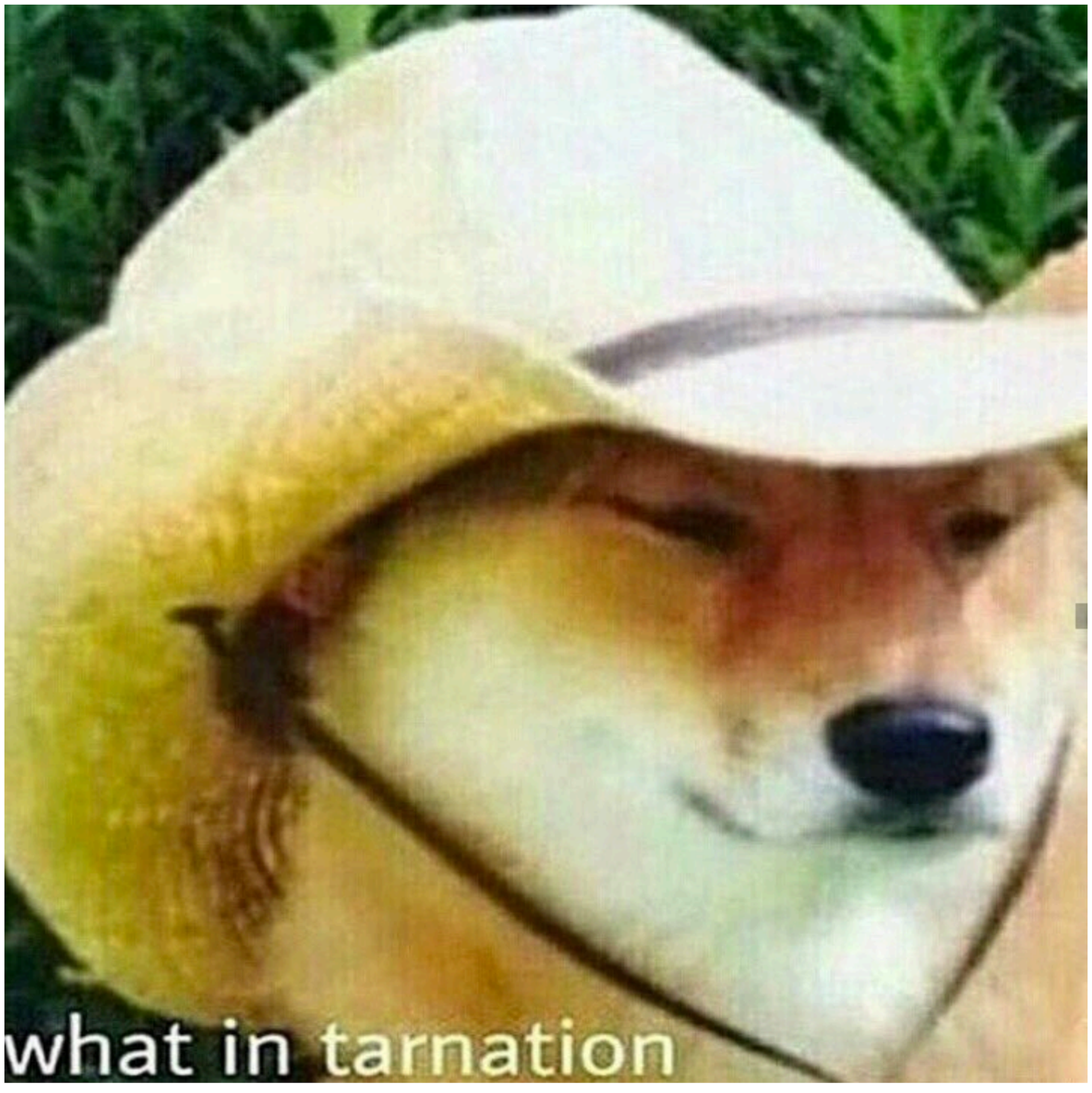
To **not move** a resource, you instead use **borrowing:** You create a reference to the resource, which can be **borrowed** by something else.

Ownership Rules

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

Reference Rules

1. At any given time, you can have either but not both of:
   - One mutable reference.
   - Any number of immutable references.
2. References must always be valid.


what in tarnation

Stack and Heap: A Review

- The stack and heap are regions of a processes memory that are used by the process at runtime
- Stack stores **known, fixed** values through pointers and can access them **very quickly**
- Heap stores data with **unknown size** at compile time or a size that **might change** and is **slower**

Stack and Heap

- Ownership addresses storing data on the heap:
  - Keeping track of what parts of code are using what data on the heap
  - Minimizing the amount of duplicate data on the heap
  - Cleaning up unused data on the heap so we don’t run out of space

Strings are mutable, and thus we don't know their size at compile-time, so we store them on the heap.

↝ **move (for types that do not implement Copy)**


```
let s = String::from("hello");
```

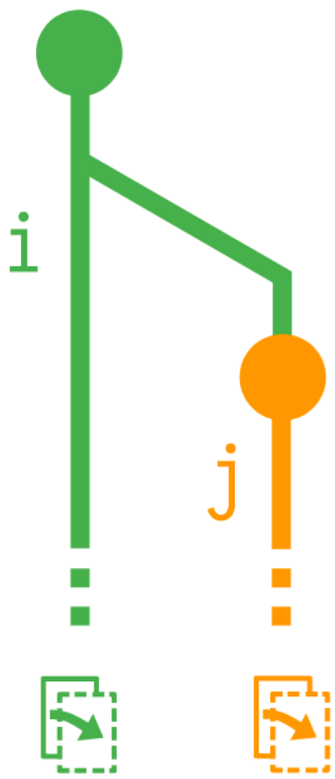
```
let b = s + " world";
```

(**cannot** use s anymore)

https://rufflewind.com/2017-02-15/rust-move-copy-borrow

Integers size is known at compile-time, and thus can be stored on the stack (i.e. no move!)

copy (for types that do implement Copy)




```
let i = 42;
```

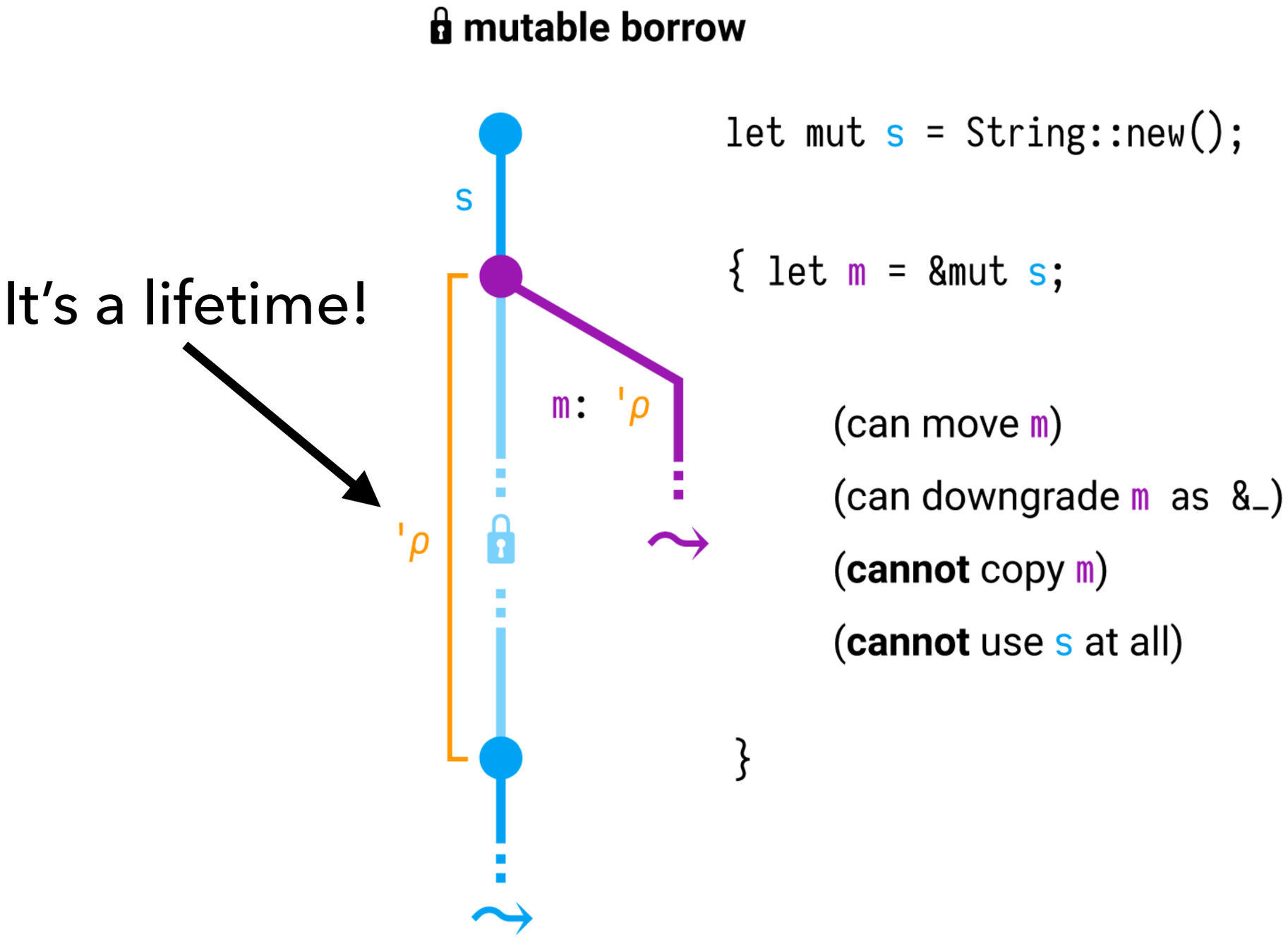
It’s a trait!

```
let j = i + 1;
```

(can still use `i` and `j`)

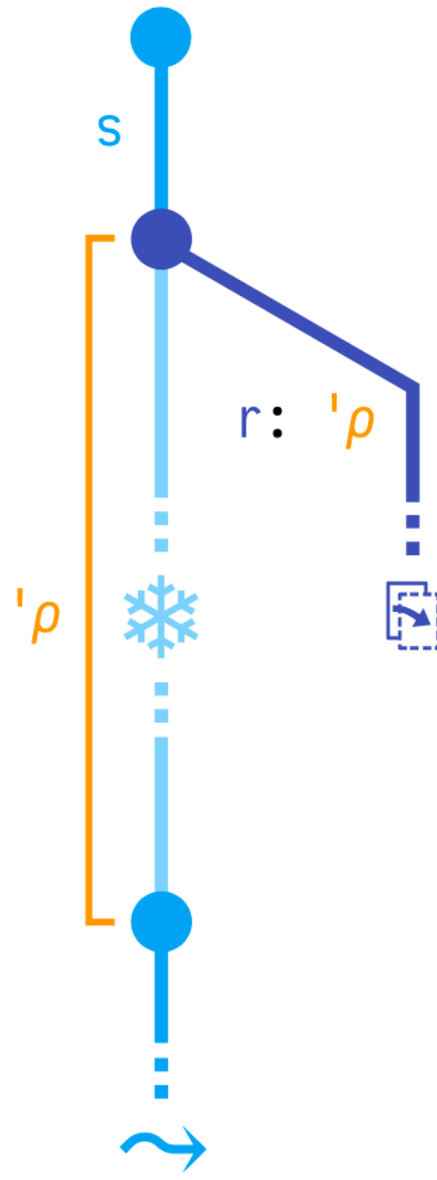
https://rufflewind.com/2017-02-15/rust-move-copy-borrow

Only one shared mutable reference at a time!



https://rufflewind.com/2017-02-15/rust-move-copy-borrow

❄ borrow




```
let s = String::from("hello");

{ let r = &s;

        (can copy r)
        (can still &s)
        (cannot &mut  s)
        (cannot move s)

}
```

https://rufflewind.com/2017-02-15/rust-move-copy-borrow

Lifetimes

Lifetime Basics

- All references in Rust have a *lifetime*
- Lifetimes are the scopes for which references are valid
- Most lifetimes are implicit and inferred
- Lifetimes eliminate **dangling references**

Lifetimes

- Consider the following program:

```
fn main() {
    let r;

    {
        let x = 5;
        r = &x;
    }

    println!("r: {}", r);
}
```

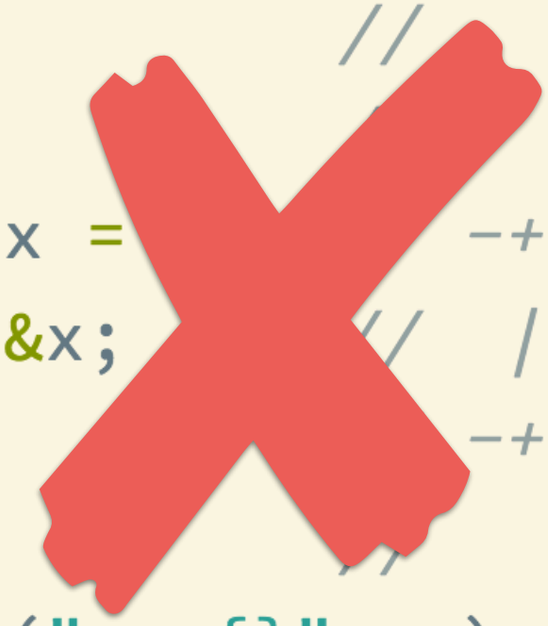
Lifetimes

- Doesn't work!

```
› cargo run
   Compiling kcdc v0.1.0 (file:///Users/jared/development/rust/projects/kcdc)
error: `x` does not live long enough
  --> src/main.rs:9:5
   |
8  |           r = &x;
   |               - borrow occurs here
9  |       }
   |       ^ `x` dropped here while still borrowed
...
12 | }
   | - borrowed value needs to live until here
```

The Borrow Checker

```
{
    let r;                // ---------+-- 'a
                          //          |
    {                     //          |
        let x =           //  -+------+-- 'b
        r = &x;           //   |      |
    }                     //  -+      |
                          //          |
    println!("r: {}", r); //  |       |
                          //          |
                          // ---------+
}
```

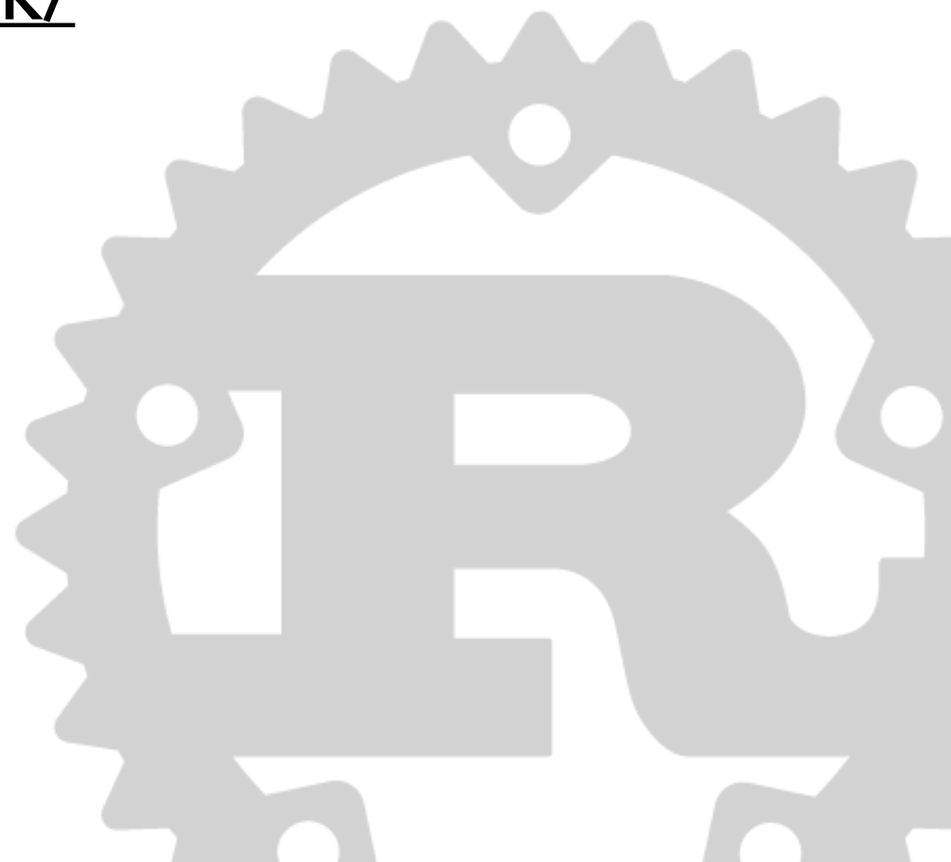


The Borrow Checker

```
{
    let x = 5;                  // -----+-- 'b
                                //      |
    let r = &x;                 // --+--+-- 'a
                                //   |  |
    println!("r: {}", r);       //   |  |
                                //   |  |
                                // --+  |
}                               // -----+
```

Slices

- Slices let you reference a contiguous sequence of elements in a collection rather than the whole collection
- We won't talk about them today, but checkout the Rust book for more on them: https://doc.rust-lang.org/book/second-edition/ch04-03-slices.html



What does this all mean?!

- In Rust, you **can't** have:
  - Segfaults
  - Use-after-free errors
  - Dangling references
  - Data races (race conditions)

@jaredthecoder | KCDC 2017

Concurrency

@jaredthecoder | KCDC 2017

Fearless
Concurrency

–A Rust Slogan

Concurrency in Rust

- Broad concurrency model
- Ownership prevents safety issues

Concurrency in Rust

- Rust includes support for:
  - Raw threads and processes
  - Message passing
  - Mutexes and Atomic Reference Counter
  - Concurrency traits for objects

Spawning Threads

```
use std::thread;

fn main() {
    let v = vec![1, 2, 3];

    let handle = thread::spawn(move || {
        println!("Here's a vector: {:?}", v);
    });

    handle.join();
}
```

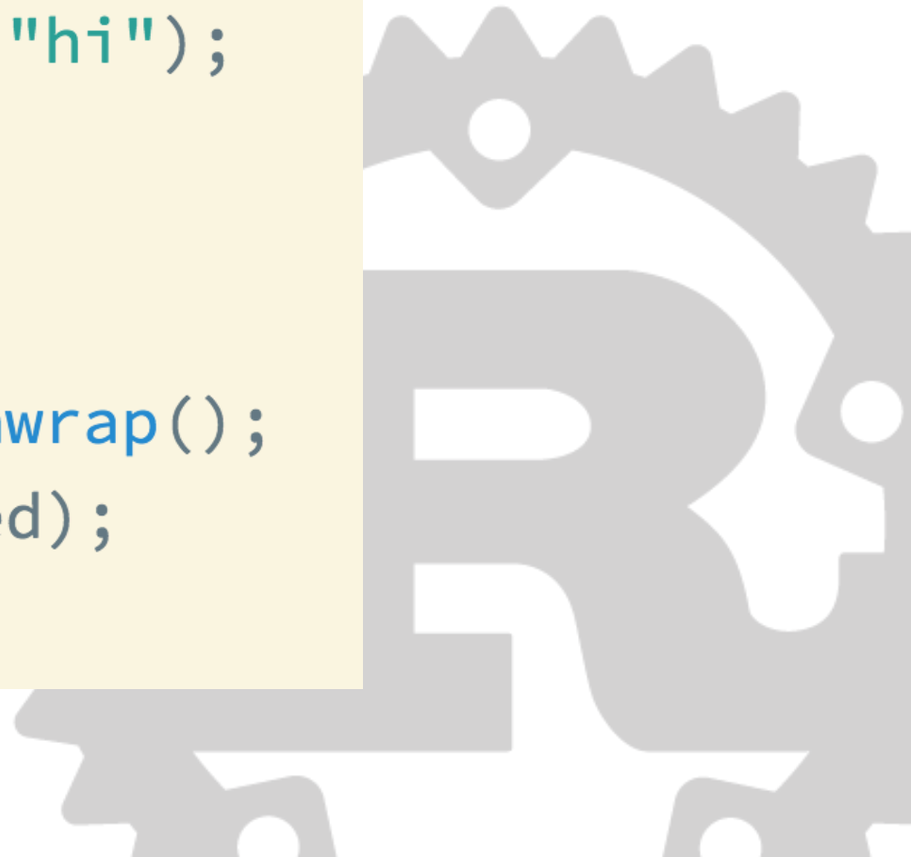
Message Passing

```
use std::thread;
use std::sync::mpsc;

fn main() {
    let (tx, rx) = mpsc::channel();

    thread::spawn(move || {
        let val = String::from("hi");
        tx.send(val).unwrap();
    });

    let received = rx.recv().unwrap();
    println!("Got: {}", received);
}
```



Sharing State

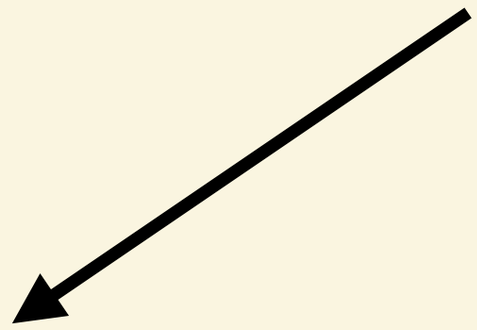
```
use std::sync::Mutex;

fn main() {
    let m = Mutex::new(5);

    {
        let mut num = m.lock().unwrap();
        *num = 6;
    }

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

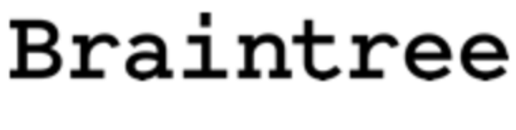
Locking is enforced by the compiler



In the Wild

It isn't just a fad.

Rust in Production



NPM

- NPM, the Node.js package manager, uses Rust to serve up over 350 million packages a day
- Used to eliminate bottlenecks in the package delivery pipeline

Source: https://www.youtube.com/watch?v=GCsxYAxw3JQ

Dropbox

- Uses Rust in multiple, high-impact projects to manage **exabytes of data** on the backend
- Switched off of AWS S3 to homegrown solution built with Rust (Go used too much memory)
- Additionally, Rust is shipped with the Dropbox desktop Windows client to **hundreds of millions** of machines

Source: https://air.mozilla.org/rust-meetup-may-2017/

Mozilla Firefox

- Using Rust in production versions of Firefox shipping to **hundreds of millions** of users
- Used in Servo, in media playback functionality, and other parts of browser
- Used in Project Quantum, a complete rewrite of the Firefox engine in Rust

Source: https://hacks.mozilla.org/2016/07/shipping-rust-in-firefox/

OneSignal

- Push notification delivery system built in Rust
- Delivers over **2 billion push notifications per week**
- Sustained deliveries of up to **125,000/second** and spikes of 175,000/second
- Powers apps at Uber, Cisco, Zillow, Adobe, and Wix

Source: https://onesignal.com/blog/rust-at-onesignal/

Wire

- Used for secure messaging
- Wire's Axolotl protocol implementation and other cryptographic and utility libraries are developed in Rust
- Cross-compiled for iOS and Android

Source: https://github.com/wireapp/proteus

Let’s write some code!

Takeaways

- Rust makes systems programming fun and easy, without compromising speed or safety.
- Try out Rust: https://play.rust-lang.org/
- Read more about Rust: https://doc.rust-lang.org/book/second-edition/

Questions?

Jared M. Smith

jaredthecoder

jaredthecoder

jaredthecoder

jaredthecoder.com

jared@jaredthecoder.com