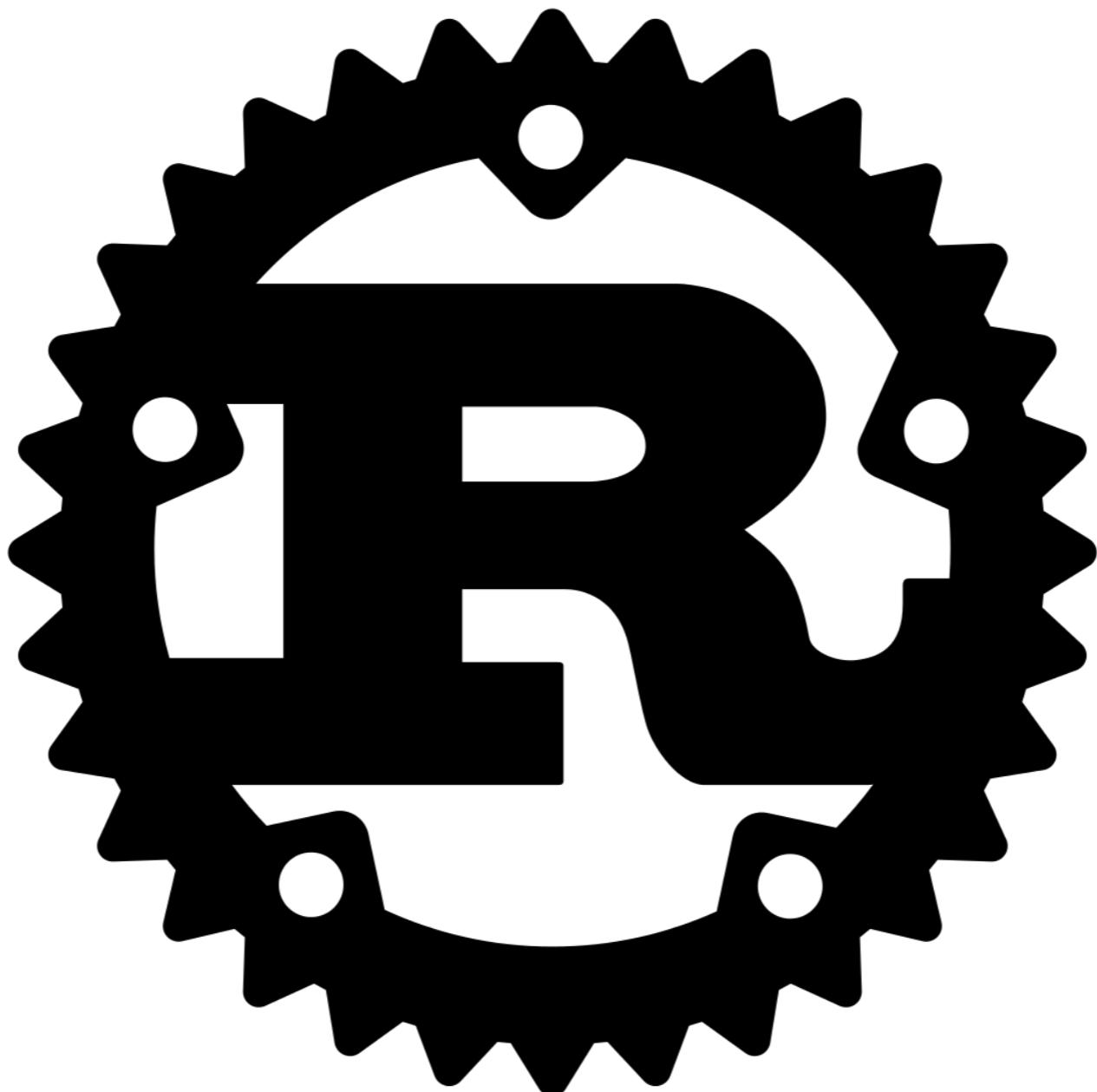


# Who got types in my Systems Programming!

Jared Roesch







*Paul G. Allen Center for Computer Science & Engineering*  
 *Opening the Doors to our Future*

PAUL G. ALLEN CENTER  
for Computer Science & Engineering

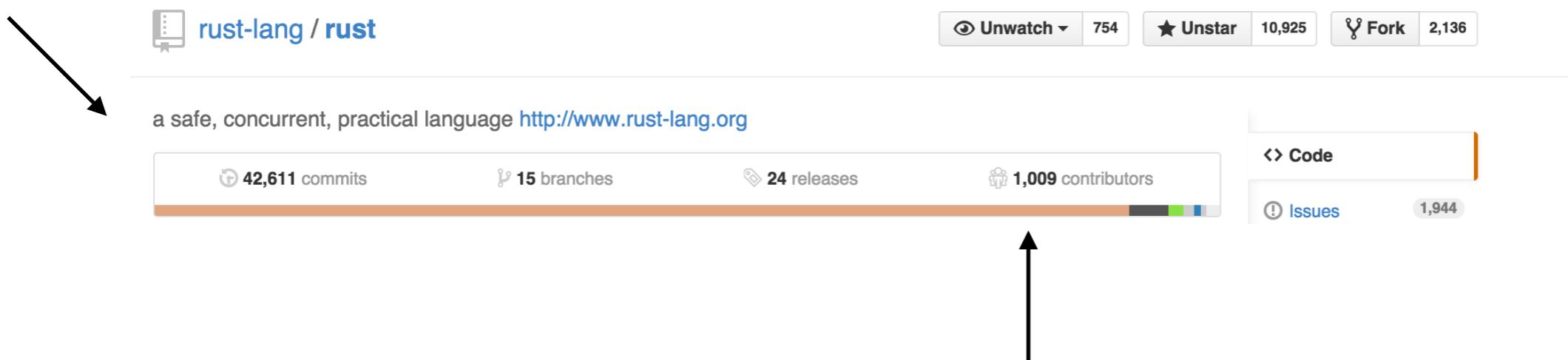
# Rust

- A new systems programming language
- 1.0 was released last week (May 15th)
- Pursuing the trifecta: safe, concurrent, fast
- Gone through many radical iterations
- Development is open source but sponsored by Mozilla

Lots of interest



Lots of commits



Lots of contributors

# Goals

- Help you understand why Rust is interesting (even if you don't use it)
- Important features of Rust and how they improve the state of systems programming
- Exercises to get your hands dirty with the language
- Show off the tooling

# Agenda

- Introduction
- Topic 1 (17 mins)
- Exercises (8 mins)
- Topic 2 (17 mins)
- Exercises (8 mins)
- Break + QA (10 mins)
- Topic 3 (17 mins)
- Exercises (8 mins)
- Topic 4 (17 mins)
- Exercises + Q&A (10 mins)

# Agenda

- **Introduction & Motivation**
- Type System Innovations
- Exercises (8 mins)
- Topic 2 (17 mins)
- Exercises (8 mins)
- Break + QA (10 mins)
- Topic 3 (17 mins)
- Exercises (8 mins)
- Topic 4 (17 mins)
- Live coding Q&A

Rust is a systems programming language that runs blazingly fast, prevents almost all crashes\*, and eliminates data races.

[Show me more!](#)

## Featuring

- zero-cost abstractions
- move semantics
- guaranteed memory safety
- threads without data races
- trait-based generics
- pattern matching
- type inference
- minimal runtime
- efficient C bindings

How? Types!

# Type Systems

- Types allow us to reason statically about program behavior
- Type checking is a form of logical reasoning
- We want to verify the consistency of a statement (program)
- Like logics, type systems can come in many flavors, some more exotic than others

# Systems Programming

- Fine grained memory control
- Zero-cost abstractions
- Pay for what you use
- Scheduling logical tasks
- Performance is a necessity
  - Usually implies "unsafe"

# Memory Control

- **Control over:**
  - Where memory is allocated (*stack vs. heap*)
  - How memory is allocated (*contiguous vs. noncontiguous*)
  - When memory is reclaimed

# Zero-cost abstraction

- We want performance + abstraction
- Need abstractions that are compiled away
- Optimization is key

# Pay for what you use

- Only incur the cost of an abstraction when you use it
- One of C++'s guiding philosophies
  - Virtual
  - Zero-cost exceptions
  - Templates

# Scheduling

- all modern systems rely on scheduling
- must be able to chunk and schedule work
- browsers, webservers, game, operating systems, etc

# It's all about Memory

- No matter what memory has a direct influence on performance
- Tension between the desire to have direct control of memory, while maintaining safety
- Traditional reasoning about memory requires knowledge of execution

# Garbage Collection

- Fully automatic; very little programmer overhead
- No control over memory layout or reclamation
- Doesn't fix the resource problem (i.e., files, sockets)
  - Non-memory resources are peril to the non-determinism of GC

# malloc + free

- Manual; lots of programmer overhead
- Explicit control over allocation, layout, and reclamation
- Ad-hoc reasoning about the lifetime and ownership of an allocation is critical to this approach; but fallible

# Memory statically

- What if we could do our reasoning about deallocations statically & automatically
- Can we encode our reasoning in a type system?
- What would it look like?

- **ownership (affinity/linearity)**
  - borrowing
  - lifetimes
  - traits

# Core Concepts

- **Ownership**
- Borrowing
- Lifetimes
- Traits

# Ownership

- Each value is owned by stack frame.
- Each use of a value "consumes it"; once consumed it is essentially no longer in scope
- In practical terms, assignment, function calls, pattern matching, etc all consume your value

```
fn main() {  
    println!("Hello World!");  
}
```

```
fn find_max(v: Vec<i32>) -> Option<&i32> {
    v.into_iter().max()
}

fn main() {
    let v = vec![4,3,2,1,5,6,10];
    println!("{:?}", find_max(v));
    println!("{:?}", v); // error: use of moved value: `v`
}
```

```
fn print_vec_with_head(v: Vec<i32>) {  
    println!("{:?}", v)  
}  
  
fn main() {  
    let v = vec![4,3,2,1,5,6,10];  
    print_vec_with_header(v);  
    // can't use v ever again  
}
```

# Core Concepts

- Ownership
- **Borrowing**
- Lifetimes
- Traits

# Borrowing

- If every operation consumes a value how do I write programs that do more than one thing?
- Borrowing allows one to "lease" data for a period of time

```
struct Vec3 { x: i32, y: i32, z: i32 }

...

fn vec_eq(v1: Vec3, v2: Vec3) -> bool {
    v1.x == v2.x &&
    v1.y == v2.y &&
    v1.z == v2.z
}

fn main() {
    let x = Vec3::new(1,2,3);
    let y = Vec3::new(3,2,1);
    let is_eq = vec_eq(x, y);
    println!("{}:?}", x); // error value moved
    println!("{}:?", y); // error value moved
}
```

This function is the problem:

```
fn vec_eq(v1: Vec3, v2: Vec3) -> bool {  
    v1.x == v2.x &&  
    v1.y == v2.y &&  
    v1.z == v2.z  
}
```

```
fn vec_eq(v1: &Vec3, v2: &Vec3) -> bool {  
    v1.x == v2.x &&  
    v1.y == v2.y &&  
    v1.z == v2.z  
}
```

```
struct Vec3 { x: i32, y: i32, z: i32 }

fn vec_eq(v1: &Vec3, v2: &Vec3) -> bool {
    v1.x == v2.x &&
    v1.y == v2.y &&
    v1.z == v2.z
}

fn main() {
    let x = Vec3::new(1,2,3);
    let y = Vec3::new(3,2,1);
    // borrow x & y
    let is_eq = vec_eq(&x, &y);
    // un-borrow x & y
    println!("{}:?}", x); // works!
    println!("{}:?", y); // works!
}
```

```
let mut x = 5;

let y = &mut x;      // --> &mut borrow of x starts here
                    // |
*y += 1;           // |
                    // |
println!("{}", x); // --> - try to borrow x here
                    // --> &mut borrow of x ends here
```

```
let mut x = 5;

{
    let y = &mut x; // -+ &mut borrow starts here
    *y += 1;        // |
}                      // -+ ... and ends here

println!("{}", x); // <- try to borrow x here
```

# Borrowing

- References have two flavors: immutable `&T` and mutable `&mut T`
- The type checker enforces that we can have any number of immutable readers, but only a single mutable writer
- Solves problems for **both** single threaded and multithreaded programs

# Core Concepts

- Ownership
- Borrowing
- **Lifetimes**
- Traits

```
fn bad_ret() -> &i32 {
    let x = 10; &x
}

fn main() {
    let x = bad_ret();
    foo_bar();
    // where does x point to?
}
```

# Lifetimes

- Memory always has a lifetime
  - Lifetimes are non-deterministic in the presence of GC
  - Lifetimes of stack variables are usually **known** but **not enforced**
  - Lifetimes of heap variables are **unknown** and **not enforced**
  - fundamental question: when will memory be freed?

```
fn example1() {
    let x = 1;                      // x is put on the stack
    let y = vec![1,2,3];            // y is put on the stack
    let z = Vec3::new(1,2,3);      // z is put on the stack
}
// entire stack frame is freed
```

```
// error: missing lifetime specifier [E0106]
fn bad_ret() -> &i32 {
    let x = 10; &x
}

fn foo_bar() { ... }

fn main() {
    let x = bad_ret();
    foo_bar();
    // where does x point to?
}
```

```
// <anon>:2:18: 2:19 error: `x` does not live long enough
// <anon>:2      let x = 10; &x

fn bad_ret<'a>() -> &'a i32 {
    let x = 10; &x
}

fn foo_bar() {}

fn main() {
    let x = bad_ret();
    foo_bar();
    // where does x point to?
}
```

more examples here?

# Checkpoint 1

- These core concepts form Rust's type system
- Keep them in mind as we explore the rest of the language
- Let's try a couple simple exercises
- Remember: ask questions at any point!

# Exercise Time



# Interlude

- Before tackling traits we will cover necessary language features
  - functions
  - datatypes
  - methods
  - loops

```
fn print_int(i : i32) {
    println!("{}", i)
}

fn print_debug<T: Debug>(t: T) {
    println!("{:?}", t);
}

fn id<A>(x: A) -> A { x }

fn abs(i: i32) {
    if i > 0 {
        return i
    } else {
        return i * -1
    }
}
```

# structs & enums

- Two ways to define data types
- Structs are products (no runtime tag)
- Enums are sums and products (runtime tag)

```
struct FileDescriptor(i32);  
  
struct Pair<A, B>(A, B);  
  
struct Vec3 {  
    x: i32,  
    y: i32,  
    z: i32  
}
```

```
enum List<A> {
    Nil,
    Cons(A, Box<List<A>>)
}

enum Tree<A> {
    Tip
    Branch(A, Box<Tree<A>>, Box<Tree<A>>)
}
```

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

# Methods

- Two flavors
  - Inherent methods
  - Trait methods
- We will look at inherent right now, trait methods a little later
- Inherent methods are syntactic sugar, and provide some namespacing

# Inherent Methods

```
impl<A> List<A> {
    fn head(&self) -> Option<&A> {
        match self {
            &Cons(ref x, ref xs) => Some(x),
            &Nil => None
        }
    }

    fn tail(&self) -> Option<&List<A>> {
        match self {
            &Nil => None,
            &Cons(_, xs) => Some(&*xs)
        }
    ...
}
```

```
impl<T> Option<T> {
    ...
    fn unwrap_or(self, default: T) -> T {
        match self {
            None => default,
            Some(x) => x
        }
    }
    ...
}
```

# Loops

- 3 types:
  - loop (while true { ... })
  - for (a loop with iterators)
  - while (arbitrary guard)

```
fn runs_forever() -> ! {
    loop {
        println!("!!!")
    }
}
```

```
let v = vec!["One", "Two", "Three"];
for c in v {
    println!("{}", c);
}
```

```
let mut sum = 0;
let mut i = 0;

let v = vec![1,3,5,7,10];

while i < v.len() {
    sum += v[i];
    i += 1;
}
```

# Recursion

- No guaranteed tail-call optimization (TCO) right now
- TCO is currently up to the LLVM optimizer
- there have been challenges with TCO and calling convention in the past
- changes to LLVM have made TCO easier, needs design and implementation work (hopefully coming soon)

# Iterators

- Lazy iteration
- Provides an efficient interface for common functional combinators
- Performance is equivalent to loops

```
fn main() {
    let iter = (0..).filter(|x| x % 2 == 0).take(5);
    for i in iter {
        println!("{}" , i)
    }
}
```

# Box

- a singly owned heap allocation
- a value of type `Box<T>` is a pointer to a value of type `T`
- the underlying `T` is deallocated when the pointer goes out of scope

```
struct BoxWrapper<T> {
    b: Box<T>
}

fn main() {
    let x = Box::new(10);
    let bw = BoxWrapper { b : x };
    // x is singly owned due to move semantics
}
```

# Errors

- Return value error handling
- No exceptions
- Panic! terminates the process
- Monadic flavor

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



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

struct Info {
    name: String,
    age: i32,
    rating: i32,
}

fn write_info(info: &Info) -> io::Result<()> {
    let mut file = try!(File::create("my_best_friends.txt"));

    try!(writeln!(&mut file, "name: {}", info.name));
    try!(writeln!(&mut file, "age: {}", info.age));
    try!(writeln!(&mut file, "rating: {}", info.rating));

    Ok(())
}
```

# Exercise Time



# Core Concepts

- Ownership
- Borrowing
- Lifetimes
- **Traits**

# Traits

- Declare abstract interfaces
- New traits can be implemented for existing types
- Old traits can be implemented for new types
- Inspired by Haskell type classes
- Enable type level programming

```
trait Show {  
    fn show(&self) -> String;  
}
```

```
impl Show for String {  
    fn show(&self) -> String {  
        self.clone()  
    }  
}
```

```
impl<A: Show> Show for List<A> {
    fn show(&self) -> String {
        match self {
            &Nil => "Nil".to_string()
            &List(ref head, ref tail) =>
                format!("{:?} :: {:?}", head, tail.show())
        }
    }
}
```

```
fn print_me<A: Show>(a: A) {  
    println!("{}", a.show())  
}
```

# `#[derive(...)]`

- Automatically implement common functionality for your types
- Let the compiler write the boilerplate for you
- Works like Haskell's deriving mechanism

```
#[derive(Debug, PartialEq, Hash, Eq, PartialOrd, Ord)]
struct User {
    name: String,
    age: i32
}

// Debug allows us to use the {:?} formatter
// PartialEq allows us to use `==`
// Eq is true equality transitive, reflexive and antisymmetric
// ...
// show example of use
User { name: "Jared", age: 22 } != User { name: "Rust", age: 1 }
```

# Uses of Traits

- As:
  - Marker traits
  - Operator Overloading; certain operations are desugared into trait methods
  - Flexible Error Handling
  - Safe Concurrency

# Marker Traits

- Copy
- Send
- Sized
- Sync

# Copy

```
#[lang="copy"]
pub trait Copy : Clone {
    // Empty.
}
```

# Clone

```
fn main() {  
    let v = vec![1,2,3];  
    let v2 = v.clone();  
    let owned_iter = v2.into_iter();  
    // v is still valid here  
}
```

# Copy

```
#[derive(Debug, Clone)]
struct MyType;

// Marker to compiler that type can be copied
impl Copy for MyType {}

fn main() {
    let x = MyType;
    let y = x; // copy occurs here
    println!("{:?}", x);
}
```

# Copy

```
#[derive(Debug, Copy, Clone)]
struct MyType;

fn main() {
    let x = MyType;
    let y = x; // copy occurs here
    println!("{:?}", x);
}
```

Send

# Sized

- by default every type is Sized
- represents types with statically known size
- needed to differentiate between dynamically sized types

# Sync

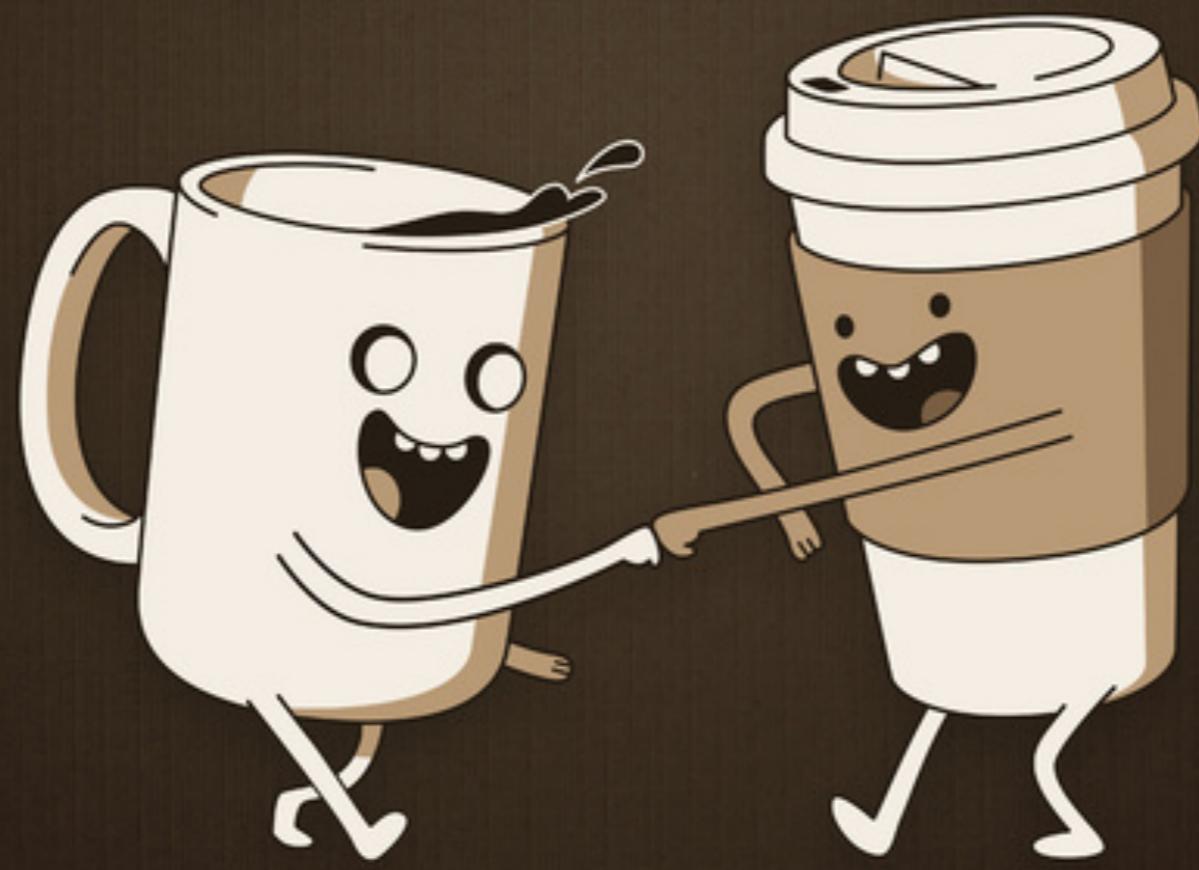
# Drop

- Allows the running of "clean-up" code
  - Close a file or socket
  - Delete a vector's underlying allocation
  - Deallocate a heap value

```
pub trait Drop {  
    fn drop(&mut self);  
}
```

```
pub struct FileDesc {  
    fd: c_int,  
}  
  
impl Drop for FileDesc {  
    ...  
    fn drop(&mut self) {  
        let _ = unsafe { libc::close(self.fd) };  
    }  
}
```

COFFEE  
TIME



# More Rust

- Associated Types
- Where Clauses
- Concurrency
- Smart Pointers
- Macros
- FFI

# Associated Types

- Allow for traits to contain type members
- Type members are abstract types
- The concrete type is selected by the implementation we select
- Enables "type level" programming

# Traits + Associated Types

- Many operations are de-sugared to trait methods:
  - $e1 + e2 \Rightarrow Add::add(e1, e2)$
  - $v[i] \Rightarrow Index::index(v, i)$
  - $*a \Rightarrow Deref::deref(a)$
- Traits with associated types allow the language be more flexible and ergonomic
  - More code in std library, less in the compiler
  - Ability to write more powerful libraries

# Operator Overloading

- Using associated types increases ergonomics
- Allows addition to have more flexible semantics
- Operations of the type `fn (A, A) -> A` don't work well in Rust; too many "similar types"
  - &u8, and u8
  - &str and String

```
pub trait Add<RHS = Self> {
    type Output;

    fn add(self, rhs: RHS) -> Self::Output;
}
```

Notice mutability here is safe,  
because we own self

```
impl<'a> Add<&'a str> for String {  
    type Output = String;  
  
    fn add(mut self, other: &str) -> String {  
        self.push_str(other);  
        self  
    }  
}
```

This allows us to be efficient by reusing the existing buffer.

# Slices

```
let v = vec![1,2,3];
println!("{:?}", v[0..]);
println!("{:?}", v[1..2]);
println!("{:?}", v[..2]);
println!("{:?}", v[2]);
```

# Index

```
pub trait Index<Idx> where Idx: ?Sized {  
    type Output: ?Sized;  
  
    fn index(&'a self, index: Idx) -> &'a Self::Output;  
}
```

```
impl<T> Index<usize> for Vec<T> {
    type Output = T;

    fn index(&self, index: usize) -> &T {
        // NB built-in indexing via `&[T]`
        &(**self)[index]
    }
}
```

```
impl<T> ops::Index<ops::RangeTo<usize>> for Vec<T> {
    type Output = [T];

    fn index(&self, index: ops::RangeTo<usize>) -> &[T] {
        Index::index(&**self, index)
    }
}
```

# Concurrency

- Send, Sync



# Rc

- a reference counted smart pointer
- implemented in the standard library
- safe interface around efficient "unsafe" operations

```
pub struct Rc<T: ?Sized> {  
    ptr :
```

# Exercise Time



# Macros

- come in two flavors:
  - compiler plugin (we won't talk about the details of these)
  - `macro_rules!`

```
macro_rules! try {
    ($expr:expr) => (match $expr {
        $crate::result::Result::Ok(val) => val,
        $crate::result::Result::Err(err) => {
            return $crate::result::Result::Err($crate::convert::From::from(err))
        }
    })
}
```

```
try!(e)

// the above call becomes:
(match e {
    Result::Ok(val) => val,
    Result::Err(err) => {
        return Result::Err(From::from(err))
    }
})
```

# FFI

- Binding to foreign code is essential
  - Can't rewrite **all** the code
  - Most system libraries are written in C/C++
  - Should allow you to encapsulate unsafe code

```
#![feature(libc)]
extern crate libc;

mod foreign {
    use libc::{c_char};

    extern {
        pub fn getchar() -> c_char;
    }
}

fn getchar() -> char {
    unsafe { std::char::from_u32(foreign::getchar() as u32).unwrap() }
}

fn main() {
    let c = getchar();
    println!("{:?}", c);
}
```

# Tooling



# Tooling

- Tooling is a critical component of when using any programming language
- Tooling can drastically impact people's perception of the language; examples:
  - sbt
  - make
  - cabal

# Testing

- Testing is baked into the Rust compiler
- Tests can be intermixed with code
- Higher level frameworks can be built on the exposed primitives
  - <https://github.com/reem/stainless>

```
#[test]
fn it_works() {
    assert!(false);
}
```

```
#[test]
#[should_panic]
fn it_works() {
    assert!(false);
}
```

```
#[test]
#[should_panic(expected = "assertion failed")]
fn it_works() {
    assert_eq!("Hello", "world");
}
```

```
describe! stainless {
    before_each {
        // Start up a test.
        let mut stainless = true;
    }

    it "makes organizing tests easy" {
        // Do the test.
        assert!(stainless);
    }

    after_each {
        // End the test.
        stainless = false;
    }

    bench "something simple" (bencher) {
        bencher.iter(|| 2 * 2)
    }
}

describe! nesting {
    it "makes it simple to categorize tests" {
        // It even generates submodules!
        assert_eq!(2, 2);
    }
}
```

```
mod stainless {
    #[test]
    fn makes_organizing_tests_easy() {
        let mut stainless = true;
        assert!(stainless);
        stainless = false;
    }

    #[bench]
    fn something_simple(bencher: &mut test::Bencher) {
        bencher.iter(|| 2 * 2)
    }
}

mod nesting {
    #[test]
    fn makes_it_simple_to_categorize_tests() {
        assert_eq!(2, 2);
    }
}
```

# cargo

## Why Cargo exists

Cargo is a tool that allows Rust projects to declare their various dependencies, and ensure that you'll always get a repeatable build.

To accomplish this goal, Cargo does four things:

- Introduces two metadata files with various bits of project information.
- Fetches and builds your project's dependencies.
- Invokes `rustc` or another build tool with the correct parameters to build your project.
- Introduces conventions, making working with Rust projects easier.

# Cargo.toml

```
[package]
name = "tower"
version = "0.1.0"
authors = ["Jared Roesch <roeschinc@gmail.com>"]
```

# Dependencies

```
[dependencies]
rustc-serialize = "*"
docopt = "*"
docopt_macros = "*"
toml = "*"
csv = "*"
threadpool = "*"
```

# Dependencies

```
[dependencies]
rustc-serialize = "0.3.14"
docopt = "*"
docopt_macros = "*"
toml = "*"
csv = "~0.14"
threadpool = "^0"
```

Pin a version

$0.14.0 < 0.15.0$

$\geq 0.0.0 < 1.0.0$

# Dependencies

```
[dependencies.color]
git = "https://github.com/bjz/color-rs"
```

# More on cargo

- Rust's SemVer: <https://github.com/rust-lang/semver>
- Cargo: <https://github.com/rust-lang/cargo>
- Crates.io: <https://crates.io/>

# rustdoc

- Documentation tool
- Completely searchable (no Hoogle equivalent yet)
- Emits static site with docs for:
  - Modules
  - Datatypes
  - Traits
  - Impls
  - etc

```
136 /**
137 /// Readers are intended to be composable with one another. Many objects
138 /// throughout the I/O and related libraries take and provide types which
139 /// implement the `Read` trait.
140 #[stable(feature = "rust1", since = "1.0.0")]
141 pub trait Read {
142     /// Pull some bytes from this source into the specified buffer, returning
143     /// how many bytes were read.
144     ///
145     /// This function does not provide any guarantees about whether it blocks
146     /// waiting for data, but if an object needs to block for a read but cannot
147     /// it will typically signal this via an `Err` return value.
148     ///
149     /// If the return value of this method is `Ok(n)`, then it must be
150     /// guaranteed that `0 <= n <= buf.len()`. A nonzero `n` value indicates
151     /// that the buffer `buf` has been filled in with `n` bytes of data from this
152     /// source. If `n` is `0`, then it can indicate one of two scenarios:
153     ///
154     /// 1. This reader has reached its "end of file" and will likely no longer
155     /// be able to produce bytes. Note that this does not mean that the
156     /// reader will *always* no longer be able to produce bytes.
157     /// 2. The buffer specified was 0 bytes in length.
158     ///
159     /// No guarantees are provided about the contents of `buf` when this
160     /// function is called, implementations cannot rely on any property of the
161     /// contents of `buf` being true. It is recommended that implementations
162     /// only write data to `buf` instead of reading its contents.
163     ///
164     ///
```

## Trait std::io::Read

[\[-\]](#) [\[+\]](#) [\[src\]](#)

std::io::Read

```
pub trait Read {
    fn read(&mut self, buf: &mut [u8]) -> Result<usize>;
    fn read_to_end(&mut self, buf: &mut Vec<u8>) -> Result<usize> { ... }
    fn read_to_string(&mut self, buf: &mut String) -> Result<usize> { ... }
    fn by_ref(&mut self) -> &mut Self where Self: Sized { ... }
    fn bytes(self) -> Bytes<Self> where Self: Sized { ... }
    fn chars(self) -> Chars<Self> where Self: Sized { ... }
    fn chain<R: Read>(self, next: R) -> Chain<Self, R> where Self: Sized { ... }
    fn take(self, limit: u64) -> Take<Self> where Self: Sized { ... }
    fn tee<W: Write>(self, out: W) -> Tee<Self, W> where Self: Sized { ... }
}
```



[\[-\]](#) A trait for objects which are byte-oriented sources.

```
[+] fn chain<R: Read>(self, next: R) -> Chain<Self, R>
  where Self: Sized
```

Creates an adaptor which will chain this stream with another.

The returned `Read` instance will first read all bytes from this object until EOF is encountered. Afterwards the output is equivalent to the output of `next`.

```
[+] fn take(self, limit: u64) -> Take<Self>
  where Self: Sized
```

Creates an adaptor which will read at most `limit` bytes from it.

This function returns a new instance of `Read` which will read at most `limit` bytes, after which it will always return EOF (`Ok(0)`). Any read errors will not count towards the number of bytes read and future calls to `read` may succeed.

## Implementors

---

```
impl Read for File
impl<'a> Read for &'a File
impl<R: Read> Read for BufReader<R>
impl<S: Read + Write> Read for BufStream<S>
impl<'a> Read for Cursor<&'a [u8]>
impl<'a> Read for Cursor<&'a mut [u8]>
impl Read for Cursor<Vec<u8>>
impl<'a, R: Read + ?Sized> Read for &'a mut R
impl<R: Read + ?Sized> Read for Box<R>
impl<'a> Read for &'a [u8]
impl Read for Empty
impl Read for Repeat
impl Read for Stdin
impl<'a> Read for StdinLock<'a>
impl<T: Read, U: Read> Read for Chain<T, U>
impl<T: Read> Read for Take<T>
impl<R: Read, W: Write> Read for Tee<R, W>
impl Read for TcpStream
impl<'a> Read for &'a TcpStream
impl Read for ChildStdout
impl Read for ChildStderr
```

## Module std::result

[[-](#)] [[+](#)] [[src](#)]

### [[-](#)] Error handling with the `Result` type

`Result<T, E>` is the type used for returning and propagating errors. It is an enum with the variants, `Ok(T)`, representing success and containing a value, and `Err(E)`, representing error and containing an error value.

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

Functions return `Result` whenever errors are expected and recoverable. In the `std` crate `Result` is most prominently used for [I/O](#).

A simple function returning `Result` might be defined and used like so:

Read



## Results for Read

std::ptr::read	Reads the value from `src` without moving it. This leaves the ...
std::io::Read	A trait for objects which are byte-oriented sources.
std::io::Read::read	Pull some bytes from this source into the specified buffer, re...
std::fs::OpenOptions::read	Sets the option for read access.
std::sync::StaticRwLock::read	Locks this rwlock with shared read access, blocking the curre..
std::sync::RwLock::read	Locks this rwlock with shared read access, blocking the curre..
std::net::Shutdown::Read	Indicates that the reading portion of this stream/socket sho...
std::slice::read	
std::boxed::Box::read	
std::fs::File::read	
std::io::Tee::read	
std::io::Take::read	
std::io::Chain::read	
std::io::BufStream::read	
std::io::Cursor::read	

# Continuing with Rust

- The Rust Programming Language: <http://doc.rust-lang.org/book/>
- #rust on <irc.mozilla.org>
- <http://www.reddit.com/r/rust>

# Acknowledgements

- The Rust community for the compiler, tooling, libraries, borrowed examples, and anything else I pointed to in my talk.
- Niko Matsakis for mentoring my work on Rust, and teaching me lots about the implementation.
- My friends @iPetesta, @aberls, @adelbertchang who helped proof read my slides and shoot ideas back and forth.

Thank you for your time! Questions?

Live coding + QA