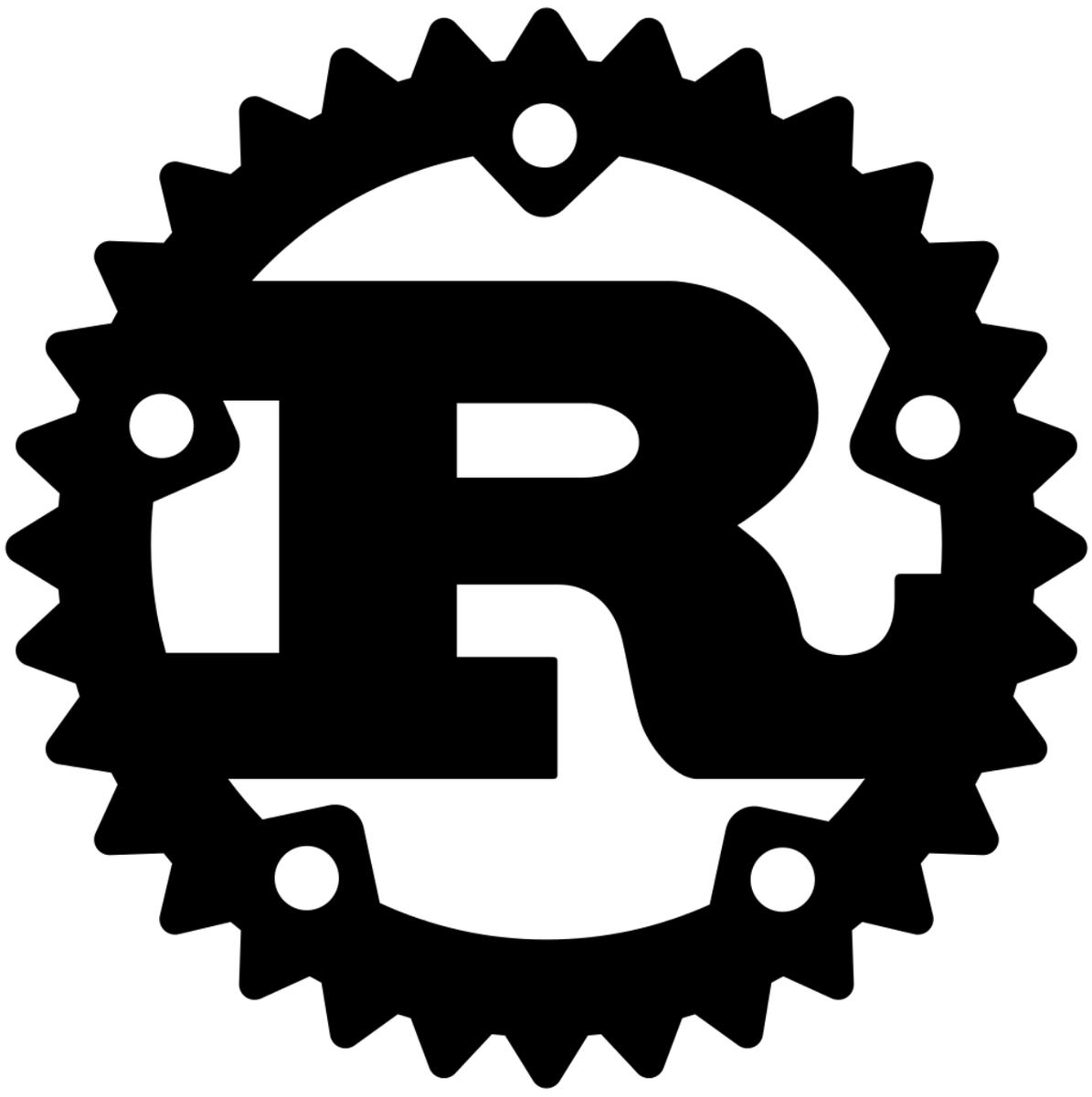
Who got types in my Systems Programming!

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 then 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
  - and more

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
- I want to have direct control of memory, but maintain 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
- lifetime and ownership of memory is one of the most critical and tricky parts

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?

The Rust type system enables this with a few core ideas:

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

Ownership

- affinity means I can "use" a value at most once
- in its theoretical form this means I can only ever "consume" a value once
- in practical terms, assignment, function calls, pattern matching, etc all consume a 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
}
```

The Rust type system consists of a couple powerful ideas:

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

Borrowing

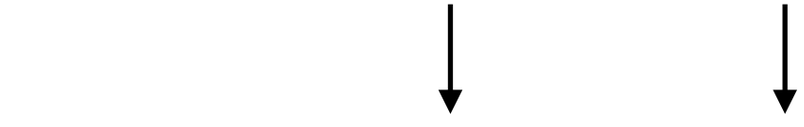
- If every operation consumes a value how do I write programs that do more than one thing?

- borrowing allows one to "lease" data for a period of time

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

...

fn vec_eq(v1: Vec3, v1: 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, v1: 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
}
```

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

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

```rust
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

The Rust type system consists of a couple powerful ideas:

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

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

```rust
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?

```rust
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
```

```rust
// 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

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

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

```rust
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
- support common functional combinators
- loop performance

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

Errors

- return value error handling
- no exceptions
- panic! terminates the process
- monadic flavor

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

```rust
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

The Rust type system consists of a couple powerful ideas:

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

Traits

- inspired by Haskell type classes
- declare abstract interfaces
- new traits can be implemented for existing types
- old traits can be implemented for new types
- enable type level programming

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

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

```rust
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(…)]`

- works like Haskell deriving
- automatically implement common functionality for your types
- let the compiler write boiler plate for you

```rust
#[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

- marker traits
- overloading; certain operations are de-sugared into trait methods
- concurrency

Special Traits Examples

- Copy
- Clone
- Send
- Sized
- Sync
- Drop

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);
}
```

Clone

```
fn main() {
    let v = vec![1,2,3];
    let v2 = v.clone();
    takes_vec(v2)
}
```

Send

Sized

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

Sync

Drop

- allows you to run "clean-up" code
- close a file or socket
- delete a vector's underlying allocation
- deallocate a heap value
- and more

```rust
pub struct FileDesc {
    fd: c_int,
}

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


COFFEE
TIME

Advanced 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
- enables "type level" programming

Traits + Associated Types

- we already know that many operations are de-sugared to trait methods
- traits with associated types allow the language be more flexible and ergonomic
    - more code in libraries
    - ability to write more powerful libraries

Operator Overloading

- using associated types increases ergonomics
- allows addition to have more flexible semantics
- operations like (a -> a -> a) don't work well in Rust
  - by reference and by value
  - slice vs. owned

```rust
pub trait Add<RHS = Self> {
    type Output;
    fn add(self, rhs: RHS) -> Self::Output;
}
```

```
impl<'a> Add<&'a str> for String {
    type Output = String;

    fn add(mut self, other: &str) -> String {
        self.push_str(other);
        self
    }
}
```

Slices

v[0..]
v[1..2]
v[..2]
v[2]
v[rat!(3/4)]

Concurrency

- Send, Sync

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

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

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

```rust
#![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);
}
```

show drop?

Tooling

Tooling

- for me a critical part of using a language is having good tooling
- tooling can drastically impact people's perception of the language
- 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

Pin a version

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

More on cargo

- Rust's SemVer: https://github.com/rust-lang/semver

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     ///
```

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 book examples, and everything else I pointed to in my talk.

Thank you for your time! Questions?

Live coding + QA