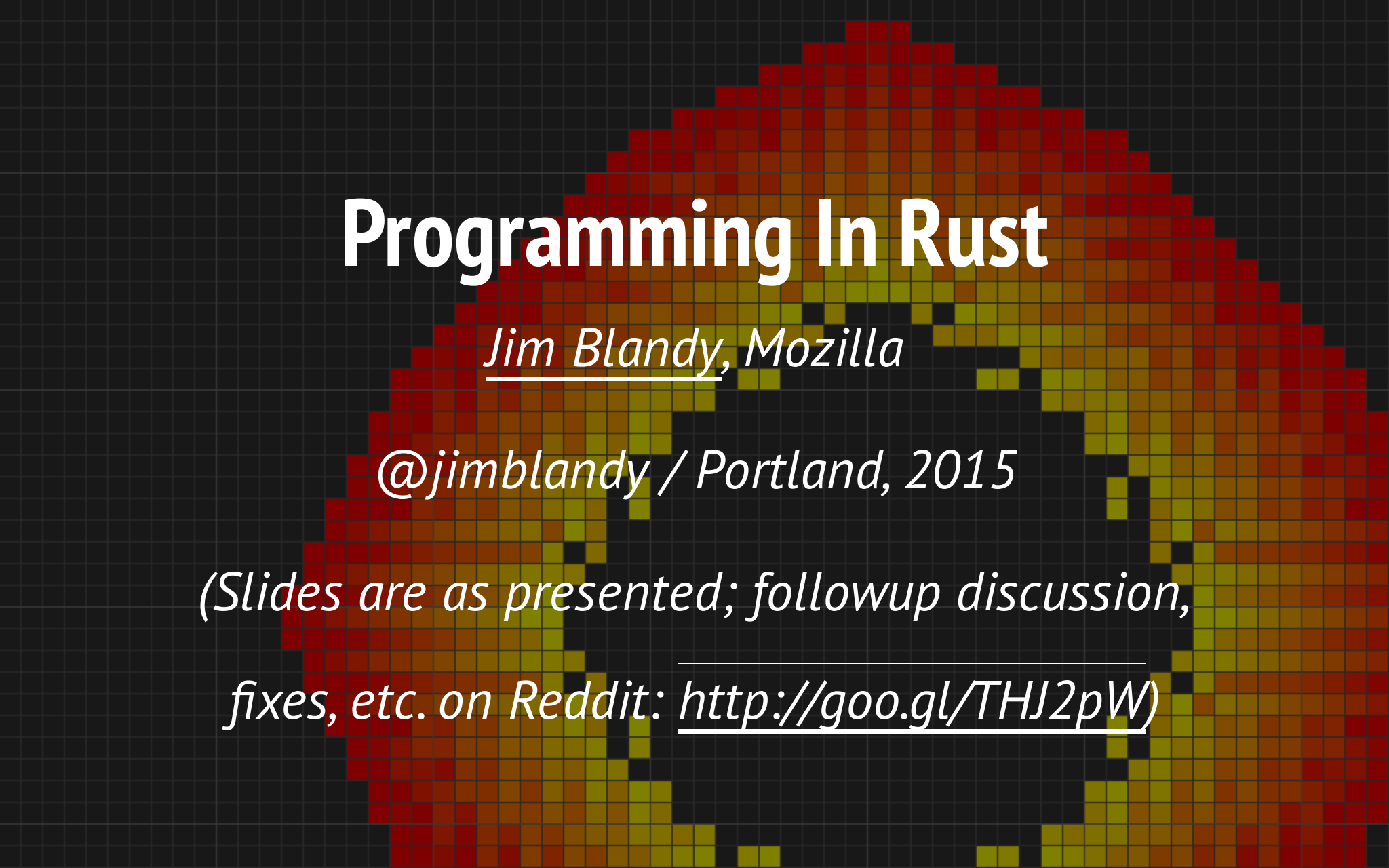


Programming In Rust

Jim Blandy, Mozilla

@jimblandy / Portland, 2015

(Slides are as presented; followup discussion, fixes, etc. on Reddit: http://goo.gl/THJ2pW)

> *The set of Rust enthusiasts certainly seems to include everyone with a Hacker News account.*
>
> ***—David Keeler***

What's the point?

Rust is a *systems programming language*:

- Language constructs have predictable performance.
- Values have predictable memory consumption.
- The language has "escape hatches" providing low-level control.

Well… *sufficiently* predictable.

What's the point?

> ***There is no GC***
>
> *A language that requires a GC is a language that opts into a larger, more complex runtime than Rust cares for. Rust is usable on bare metal with no extra runtime. Additionally, garbage collection is frequently a source of non-deterministic behavior. Rust provides the tools to make using a GC possible*
>
> ...
>
> —Rust Design FAQ

What's the point?

> *Memory safety must never be compromised.*
>
> —Rust Design FAQ

What's the point?

Memory safety means that memory is used according to its type:

- No dangling pointers.
- No leaks.
- No null pointer dereferences.
- No buffer overruns.

Rust catches the first three at compile time!

What's the point?

Memory safety has big implications for multi-threaded code.

- Threads never share mutable values directly.
- Communication occurs through primitives designed for the purpose.
- Non-deterministic behavior is localized.

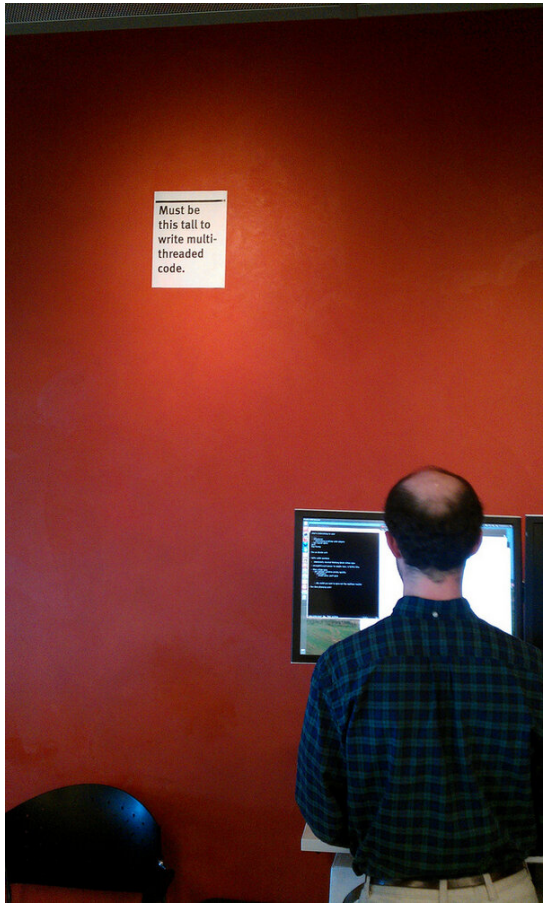


photo ©David Herman (@littlecalculist), used with permission

What's the point?

Well, never say "never". Rust has:

- raw pointers
- a C foreign function interface
- inline assembly

But all these are restricted to `unsafe` blocks. If you avoid those, you can trust the compiler's imprimatur.

Cargo's Hello, World

Cargo's Hello, World

```
sergei:rust$ cargo new --bin hello
sergei:rust$ cd hello
sergei:hello$ ls -la
total 24
drwxrwxr-x.  4 jimb jimb 4096 Jan 19 10:22 .
drwx------. 24 jimb jimb 4096 Jan 19 10:22 ..
-rw-------.  1 jimb jimb   89 Jan 19 10:22 Cargo.toml
drwxrwxr-x.  6 jimb jimb 4096 Jan 19 10:22 .git
-rw-------.  1 jimb jimb    8 Jan 19 10:22 .gitignore
drwx------.  2 jimb jimb 4096 Jan 19 10:22 src
sergei:hello$ ls src
main.rs
sergei:hello$ cat src/main.rs
fn main() {
    println!("Hello, world!");
}
sergei:hello$
```

Cargo's Hello, World

```
sergei:hello$ cat src/main.rs
fn main() {
    println!("Hello, world!");
}
sergei:hello$ cargo run
   Compiling hello v0.0.1 (file:///home/jimb/rust/hello)
     Running `target/hello`
Hello, world!
sergei:hello$ cargo clean
sergei:hello$
```

Syntax

Syntax

C++	Rust
`if (`E`)` S_1 `else` S_2	`if` E `{` S_1 `} else {` S_2 `}`
`while (`E`)` S	`while` E `{` S `}`
`for (`E_1`;` E_2`;` E_3`)` S	`for` V `in` E_{iter} `{` S `}`
`for (;;)` S	`loop {` S `}`
`int32_t i[2] = {7,11}`	`let i : [i32; 2] = [7,11]`

Syntax

C++	Rust

`int parse(char *text)`
`{`
`  S; ...`
`}`
	`fn parse(text : &str) -> i32`
	`{`
	`  E`
	`}`

Syntax

C++	Rust
`(`E_1 `&` E_2`) ==` E_3	E_1 `&` E_2 `==` E_3
`(`E_1 `|` E_2`) ==` E_3	E_1 `|` E_2 `==` E_3
E_1 `||` E_2	E_1 `||` E_2
E_1 `&&` E_2	E_1 `&&` E_2

Syntax

C++	Rust
`E.M`	`E.M`
`E->M`	`E.M`

Syntax

C++	Rust
E_1 ? E_2 : E_3	if E_1 { E_2 } else { E_3 }

Syntax

```
fn gcd(n: u64, m: u64) -> u64 {
    assert!(n != 0 && m != 0);
    if n > m {
        gcd(n - m, m)
    } else if n < m {
        gcd(m - n, n)
    } else {
        n
    }
}
```

Syntax

```
fn gcd(mut m: u64, mut n: u64) -> u64 {
    assert!(m != 0 && n != 0);
    while m != 0 {
        if m < n {
            let t = m; m = n; n = t;
        }
        m = m % n;
    }
    n
}
```

Types

Types

Primitive types:

`i8` , `i16` , `i32` , `i64`	fixed-width signed integers
`u8` , `u16` , `u32` , `u64`	fixed-width unsigned integers
`f32` , `f64`	floating-point types
`isize` , `usize`	address-sized integers
`char`	Unicode scalar value (32 bits)
`bool`	Boolean values

Types

Literal	Type
`3`	*any integral type (inferred)*
`3.`	*any floating-point type (inferred)*
`42i8`, `1729u64`	`i8`, `u64`
`-64is`, `200us`	`isize`, `usize`
`'H'`, `b'H'`	`char` (and thus Unicode), `u8`

Types

Owning types

[*T*; *N*]	fixed-size array of *T*
Vec<*T*>	growable vector of *T*
String	growable UTF-8 string
std::collections::HashMap<*K*, *V*>	map from *K* to *V*
Box<*T*>	owning pointer to *T*

Types

Borrowed pointer types (never null):

&*T*	immutable reference to *T*
&mut *T*	mutable reference to *T*
&[*T*]	slice (pointer with length) of *T*
&mut [*T*]	mutable slice of *T*
&str	slice of UTF-8 string (always immutable)

Borrowing

Borrowing

Rust either:

- proves at compile-time that the referent outlives the borrow; or
- rejects your program.

Borrowing

Three rules of borrowing:

- Either **one mutable borrow**, or **any number of immutable borrows** at a time.
- No changing values while immutably borrowed.
- No using values at all while mutably borrowed.

Borrowing

```
let mut a = 31is;
let p1 = &a;
let p2 = &a;
assert_eq!(*p1, *p2);
assert_eq!(a+1, 32);
```

everything is splendid

Borrowing

```
let mut a = 31is;
let p1 = &a;
a += 1;
```

```
error: cannot assign to `a` because it is borrowed
     a += 1;
     ^~~~~~
note: borrow of `a` occurs here
     let p1 = &a;
                ^
```

Borrowing

```
let mut a = 31is;

let p1 = &mut a;
*p1 += 1;
```

peachy

Borrowing

```
let mut a = 31is;

let p1 = &mut a;
*p1 += 1;

a;
```

```
error: cannot use `a` because it was mutably borrowed
```

Borrowing

```
let mut a = 31is;
{
    let p1 = &mut a;
    *p1 += 1;
}
a;
```

all is forgiven

Borrowing

```
let mut a = 31is;

change_it(&mut a);

a;
```

function calls are like blocks

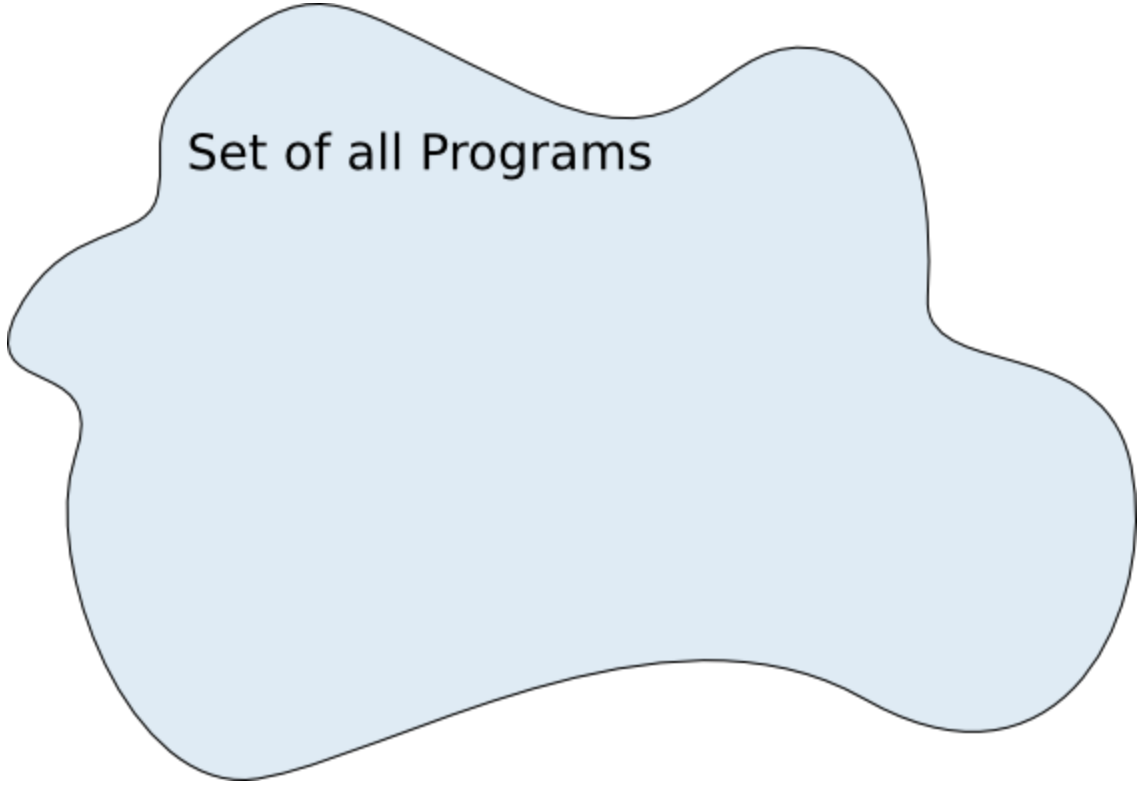
Borrowing

What is the lifetime of a borrow?

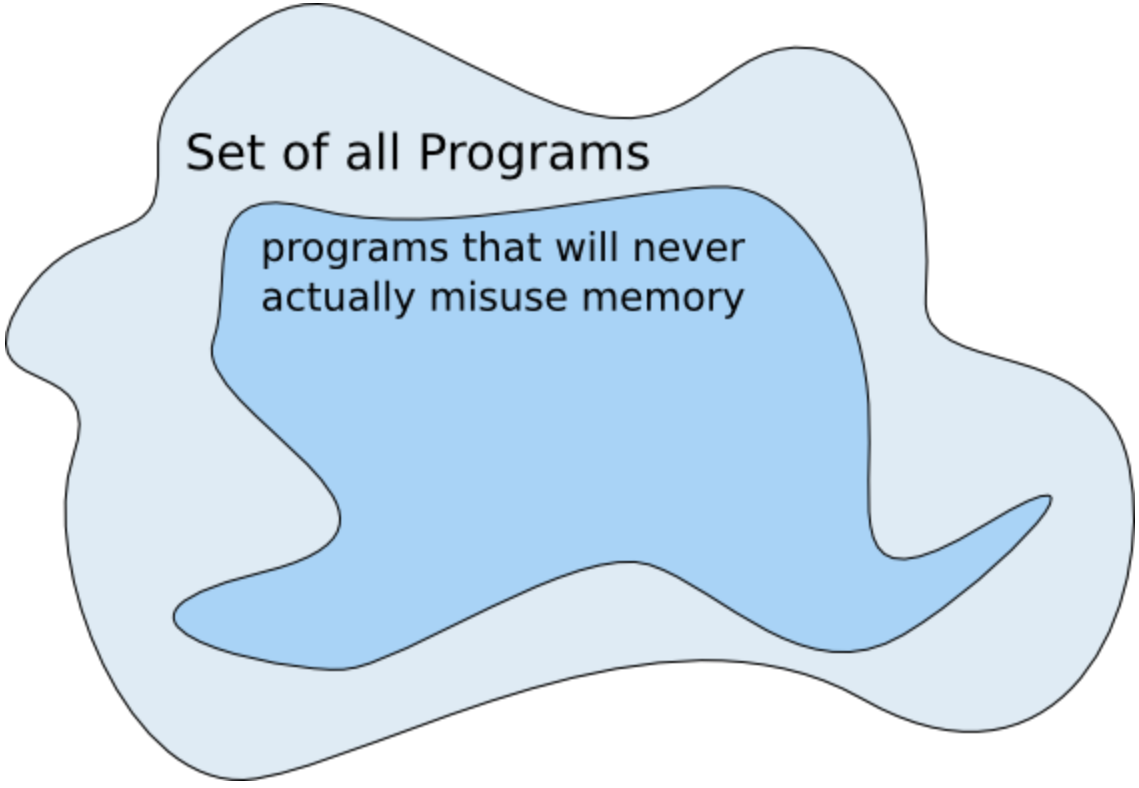
- A borrow stored in a variable extends through the variable's scope.
- A borrow passed to a function:
  - ... is confined to the call, if the function returns no references.
  - may survive the call, otherwise.

The borrow checker may change over time, expanding the set of acceptable programs.

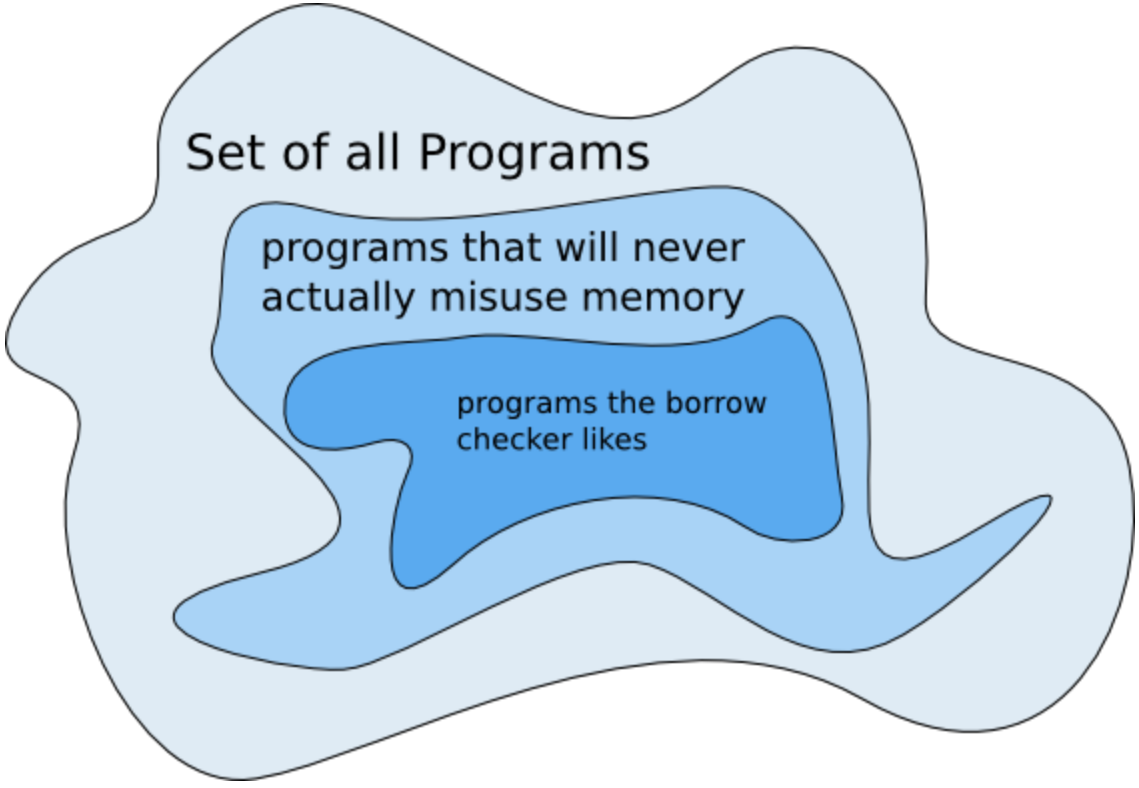
Borrowing



Borrowing


Set of all Programs
programs that will never
actually misuse memory

Borrowing


Set of all Programs
programs that will never actually misuse memory
programs the borrow checker likes

Moving

Moving

```
let mut s = String::new();
s.push_str("Hello, world!");
assert_eq!(s, "Hello, world!");
```

Moving

```
fn build(mut s: String) {
    s.push_str("Hello, world!");
}

let mut g = String::new();
build(g);
assert_eq!(g, "Hello, world!");
```

```
error: use of moved value: `g`
note: `g` moved here because it has type
`collections::string::String`, which is moved
```

Moving

Some types are *moved* by the assignment operator, and when passed to a function by value.

When moved, the destination takes ownership. The source is dead.

Such types can be passed by reference; that doesn't move them.

Moving

Which types move, and which copy?

Roughly speaking: if a simple bit-copy is adequate to copy the value, then it doesn't move; it gets copied.

So primitive numeric types, and structures containing only such, copy.

Strings, Vecs, and Boxes all containing owning pointers to memory; a full copy requires more than just a bit copy. So they move.

Moving

```
fn build(mut s: String) {
    s.push_str("Hello, world!");
}

let mut g = String::new();
build(g);
assert_eq!(g, "Hello, world!");
```

```
error: use of moved value: `g`
note: `g` moved here because it has type
`collections::string::String`, which is moved
```

Moving

```
fn build(s: &mut String) {
    s.push_str("Hello, world!");
}

let mut g = String::new();
build(&mut g);
assert_eq!(g, "Hello, world!");
```

pass by mutable reference, and all is forgiven

Moving

```
let g = "Hello".to_string();
if specific_recipient {
    append_name(g);
} else {
    append_world(g);
}
```

totally fine; only one move one each path

Moving

```
let mut g = String::new();
for _ in range(0us, 16) {
    append_something(g);
}
```

rejected; move checker knows about loops

More Types

More Types

	type	**literal form**
tuple	`(`*T_1*`, `*T_2*`, ...)`	`( 1729, "Cubes" )`

named structure type	`struct` *Name*
	`{`
	`  `*V* `:` *T*`, ...`
	`}`
		Name `{` *V*`:` *E*`, ... }`

More Types

C++	Rust

`template<P>`
`struct name`
`{`
`  T decl, ...;`
`  ...`
`}`
	`struct name<P>`
	`{`
	`  V : T,`
	`  ...`
	`}`

More Types

C++

```
enum Cpu {
  X86,
  X86_64,
  ARM
};
```

Rust

```
enum Cpu {
  X86,
  X86_64,
  ARM
}
```

More Types

```
enum ParseCoordsResult {
    Coords(f64, f64),
    Error
}


fn parse_coords(s: &str) -> ParseCoordsResult {
    ...
    if s doesn't parse well {
        return ParseCoordsResult::Error;
    }
    ...
    ParseCoordsResult::Coords(x, y)
}
```

More Types

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

fn to_i32(...) -> Option<i32>
```

More Types

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

Traits

Traits

```
trait ToString {
    fn to_string(&self) -> String;
}


impl Trait for Type {
    fn ...
}
```

Traits

```
struct Coords { x: f64, y: f64 }
```

Traits

```
struct Coords { x: f64, y: f64 }

impl ToString for Coords {
    fn to_string(&self) -> String {
        format!("({}, {})", self.x, self.y)
    }
}
```

Traits

```
struct Coords { x: f64, y: f64 }

impl ToString for Coords {
    fn to_string(&self) -> String {
        format!("({}, {})", self.x, self.y)
    }
}

#[test]
fn test_coords_to_string() {
    let pt = Coords { x: 3., y: 4. };
    assert_eq!(pt.to_string(), "(3, 4)");
}
```

Traits

```
trait Show {
    fn fmt(&self, &mut Formatter) -> Result<(), Error>;
}
```

Traits

```
struct Coords { x: f64, y: f64 }

impl Show for Coords {
    fn fmt(&self, f: &mut Formatter) -> Result<(), Error> {
        write!(f, "({}, {})", self.x, self.y)
    }
}

#[test]
fn test_format_coords() {
    let pt = Coords { x: 3., y: 4. };
    assert_eq!(format!("{:?}", pt), "(3, 4)");
}
```

Traits

```
#[derive(Show)]
struct Coords { x: f64, y: f64 }

#[test]
fn test_format_coords() {
    let pt = Coords { x: 3., y: 4. };
    assert_eq!(format!("{:?}", pt),
               "Coords { x: 3f64, y: 4f64 }");
}
```

Traits

```
#[derive(Show)]
struct Coords { x: f64, y: f64 }

impl Add for Coords {
    type Output = Coords;
    fn add(self, rhs: Coords) -> Coords {
        Coords { x: self.x + rhs.x,
                 y: self.y + rhs.y }
    }
}
```

Traits

```
#[derive(Show)]
struct Coords { x: f64, y: f64 }

impl Add for Coords { ... }

#[test]
fn test_add_coords() {
    let p1 = Coords { x: 1., y: 2. };
    let p2 = Coords { x: 4., y: 8. };
    assert_eq!(p1 + p2, Coords { x: 5., y: 10. });
}
```

fails: no equality

Traits

```
#[derive(Show, PartialEq)]
struct Coords { x: f64, y: f64 }

impl Add for Coords { ... }

#[test]
fn test_add_coords() {
    let p1 = Coords { x: 1., y: 2. };
    let p2 = Coords { x: 4., y: 8. };
    assert_eq!(p1 + p2, Coords { x: 5., y: 10. });
}
```

"You rang, sir?"

Traits

```
trait Iterator<A> {
    fn next(&mut self) -> Option<A>;
    fn size_hint(&self) -> (usize, Option<usize>);
}
```

The latest definition of Iterator is distractingly hairier.

Traits and Generics

Traits and Generics

```
fn gcd(mut m: u64, mut n: u64) -> u64 {
    assert!(m != 0 && n != 0);
    while m != 0 {
        if m < n {
            let t = m; m = n; n = t;
        }
        m = m % n;
    }
    n
}
```

Unsatisfying! Why u64?

Traits and Generics

```
fn gcd<T>(mut m: T, mut n: T) -> T {
    assert!(m != 0 && n != 0);
    while m != 0 {
        if m < n {
            let t = m; m = n; n = t;
        }
        m = m % n;
    }
    n
}
```

```
error: binary operation `!=` cannot be applied to type `T`
error: binary operation `<` cannot be applied to type `T`
error: binary operation `%` cannot be applied to type `T`
```

Traits and Generics

```
use std::num::Int;
fn gcd<T: Int>(mut m: T, mut n: T) -> T {
    assert!(m != Int::zero() && n != Int::zero());
    while m != Int::zero() {
        if m < n {
            let t = m; m = n; n = t;
        }
        m = m % n;
    }
    n
}
```

```
use std::collections::BTreeMap;
use std::io;

fn main() {
    let mut counts = BTreeMap::new();
    for line in io::stdin().lock().lines() {
        let mut line = line.unwrap();
        line.pop();
        let count = match counts.get(&line) {
            Some(v) => *v,
            None => 0us
        };
        counts.insert(line, count + 1);
    }

    for (line, count) in counts.iter() {
        println!("{} {}", count, line);
    }
}
```

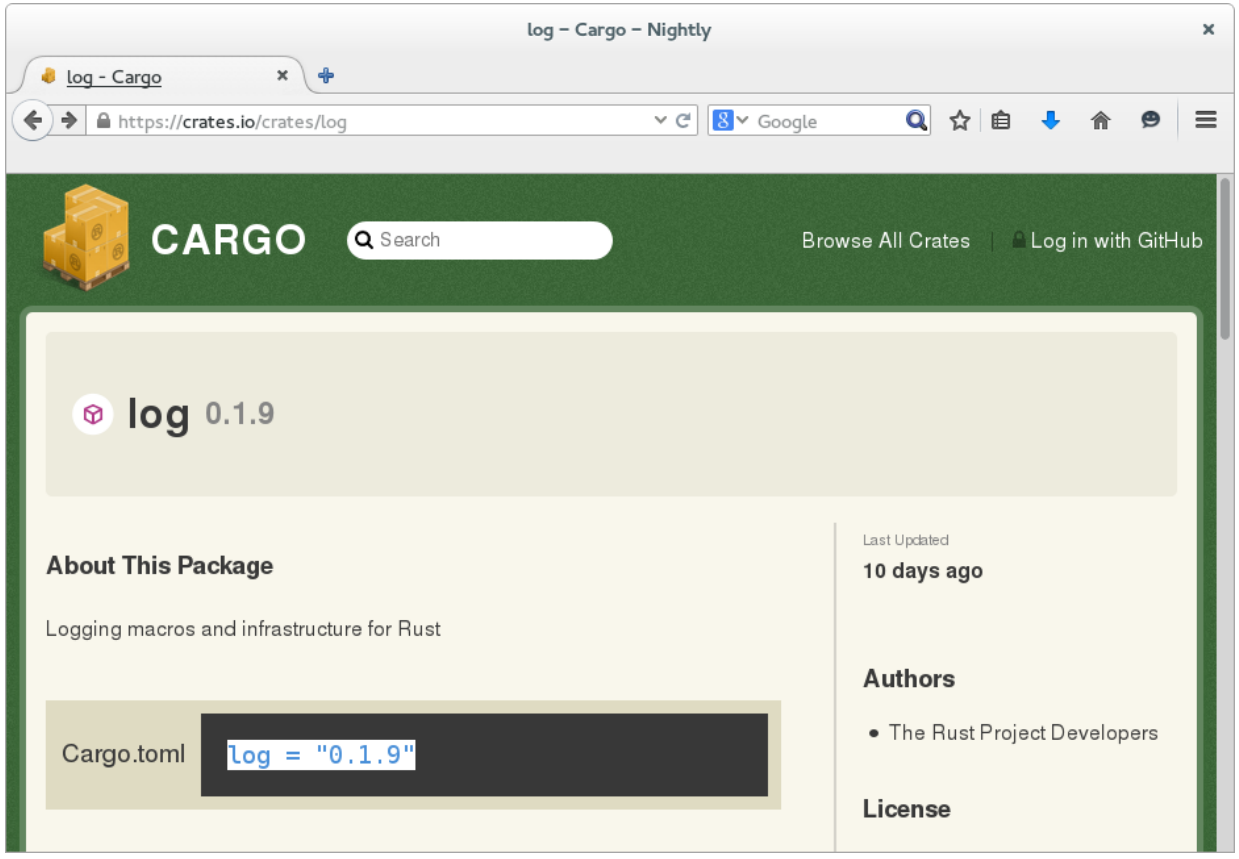
Crates, Modules, and Cargo

Crates, Modules, and Cargo

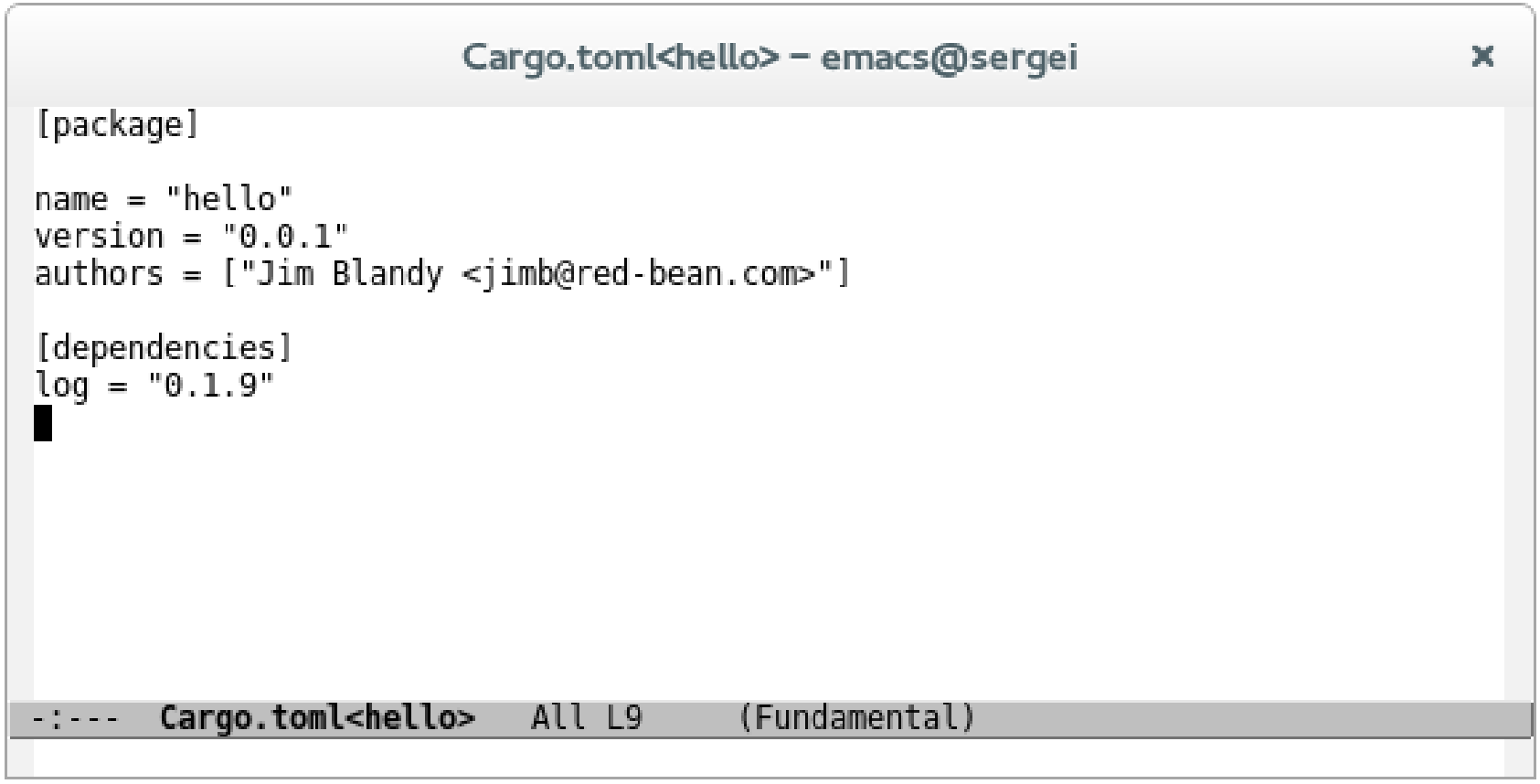
```
sergei:hello$ ls -la
total 28
drwxrwxr-x.  4 jimb jimb 4096 Jan 19 10:32 .
drwx------. 24 jimb jimb 4096 Jan 19 10:22 ..
-rw-------.  1 jimb jimb   41 Jan 19 10:32 Cargo.lock
-rw-------.  1 jimb jimb   89 Jan 19 10:22 Cargo.toml
drwxrwxr-x.  6 jimb jimb 4096 Jan 19 10:22 .git
-rw-------.  1 jimb jimb    8 Jan 19 10:22 .gitignore
drwx------.  2 jimb jimb 4096 Jan 19 10:22 src
sergei:hello$ cat Cargo.toml
[package]

name = "hello"
version = "0.0.1"
authors = ["Jim Blandy <jimb@red-bean.com>"]
sergei:hello$
```

Crates, Modules, and Cargo



Crates, Modules, and Cargo



Crates, Modules, and Cargo

main.rs<hello> – emacs@sergei

```
#[macro_use]
extern crate log;

fn main() {
    println!("Hello, world!");
    debug!("greeting level: {}", "indiscriminate");
}
```

```
-:---  main.rs<hello>   All L6     (Rust)
Wrote /home/jimb/rust/hello/src/main.rs
```

Crates, Modules, and Cargo

```
sergei:hello$ cargo run
    Updating registry `https://github.com/rust-lang/crates.io-index`
   Compiling regex v0.1.10
   Compiling log v0.1.9
   Compiling hello v0.0.1 (file:///home/jimb/rust/hello)
     Running `target/hello`
Hello, world!
sergei:hello$ RUST_LOG=debug ./target/hello
Hello, world!
DEBUG:hello: greeting level: indiscriminate
sergei:hello$
```

Crates, Modules, and Cargo

Crates, Modules, and Cargo

```
fn main() {
    println!("Hello, world!");
    println!("Read a file: {}",
             debug_info::dwarf::read().kind());
    println!("Read a file: {}",
             debug_info::stabs::read().kind());
}

mod debug_info { ... }
```

```
mod debug_info {
    trait DebugInfo {
        fn kind(&self) -> &'static str;
    }

    mod dwarf {
        struct Dwarf;
        impl DebugInfo for Dwarf {
            fn kind(&self) -> &'static str { "DWARF" }
        }
        fn read() -> Dwarf { Dwarf }
    }

    mod stabs {
        struct Stabs;
        impl DebugInfo for Stabs {
            fn kind(&self) -> &'static str { "STABS" }
        }
        fn read() -> Stabs { Stabs }
    }
}
```

```
mod debug_info {
    trait DebugInfo {
        fn kind(&self) -> &'static str;
    }

    mod dwarf {
        struct Dwarf;
        impl super::DebugInfo for Dwarf {
            fn kind(&self) -> &'static str { "DWARF" }
        }
        fn read() -> Dwarf { Dwarf }
    }

    mod stabs {
        struct Stabs;
        impl super::DebugInfo for Stabs {
            fn kind(&self) -> &'static str { "STABS" }
        }
        fn read() -> Stabs { Stabs }
    }
}
```

Crates, Modules, and Cargo

```
use debug_info::DebugInfo;

fn main() {
    println!("Hello, world!");
    println!("Read a file: {}",
             debug_info::dwarf::read().kind());
    println!("Read a file: {}",
             debug_info::stabs::read().kind());
}

mod debug_info { ... }
```

```
mod debug_info {
    pub trait DebugInfo {
        fn kind(&self) -> &'static str;
    }

    pub mod dwarf {
        struct Dwarf;
        impl super::DebugInfo for Dwarf {
            fn kind(&self) -> &'static str { "DWARF" }
        }
        pub fn read() -> Dwarf { Dwarf }
    }

    pub mod stabs {
        struct Stabs;
        impl super::DebugInfo for Stabs {
            fn kind(&self) -> &'static str { "STABS" }
        }
        pub fn read() -> Stabs { Stabs }
    }
}
```

Crates, Modules, and Cargo

```
sergei:mods$ cargo run
   Compiling mods v0.0.1 (file:///home/jimb/rust/mods)
     Running `target/mods`
Hello, world!
Read a file: DWARF
Read a file: STABS
sergei:mods$
```

Crates, Modules, and Cargo

There are three forms of the **mod** declaration:

syntax	**meaning**
`mod N { items }`	in-place: structuring code within a file
`mod N;`	source located elsewhere
—source file named *N*.rs	code comes from *N*.rs
—subdirectory named *N*	code comes from *N*/mod.rs

Crates, Modules, and Cargo

```
mod debug_info {
    pub trait DebugInfo {
        fn kind(&self) -> &'static str;
    }

    pub mod dwarf;
    pub mod stabs;
}
```

Crates, Modules, and Cargo

```
sergei:mods$ ls -lR src
src:
total 8
drwxrwxr-x. 2 jimb jimb 4096 Jan 19 11:54 debug_info
-rw-------. 1 jimb jimb  389 Jan 19 11:54 main.rs

src/debug_info:
total 8
-rw-rw-r--. 1 jimb jimb 130 Jan 19 11:52 dwarf.rs
-rw-rw-r--. 1 jimb jimb 130 Jan 19 11:54 stabs.rs
sergei:mods$
```

Crates, Modules, and Cargo

```
use debug_info::DebugInfo;

fn main() {
    println!("Hello, world!");
    println!("Read a file: {}",
             debug_info::dwarf::read().kind());
    println!("Read a file: {}",
             debug_info::stabs::read().kind());
}

mod debug_info;
```

Crates, Modules, and Cargo

```
sergei:mods$ ls -lR src
src:
total 8
drwxrwxr-x. 2 jimb jimb 4096 Jan 19 12:05 debug_info
-rw-------. 1 jimb jimb  275 Jan 19 12:05 main.rs

src/debug_info:
total 12
-rw-rw-r--. 1 jimb jimb 130 Jan 19 11:52 dwarf.rs
-rw-rw-r--. 1 jimb jimb  91 Jan 19 12:05 mod.rs
-rw-rw-r--. 1 jimb jimb 130 Jan 19 11:54 stabs.rs
sergei:mods$
```

Crates, Modules, and Cargo

The contents of `src/debug_info/mod.rs`:

```
pub trait DebugInfo {
    fn kind(&self) -> &'static str;
}

pub mod dwarf;
pub mod stabs;
```

Threads

Threads

A **scoped** thread, when joined, returns its closure's result:

```
assert!(Thread::scoped(|| { gcd(5610, 57057) })
        .join().ok() == Some(33));
assert!(Thread::scoped(|| { gcd(0, 10) })
        .join().is_err());
assert!(Thread::scoped(|| { gcd(10, 0) })
        .join().is_err());
```

Threads

A **channel** carries values between threads.

```
let (tx, rx) = std::sync::mpsc::channel();
let thread = Thread::scoped(move || {
    tx.send(127is).unwrap();
});
assert_eq!(rx.recv().unwrap(), 127);
assert!(thread.join().is_ok());
```

Brace Yourselves

```
use std::collections::BTreeMap;
use std::io::{FileStat, FileType, IoResult};
use std::io::fs::{PathExtensions, readdir};
use std::path::posix::Path;
use std::sync::mpsc::{channel, Sender};
use std::sync::TaskPool;

pub type StatMap = BTreeMap<Path, FileStat>;

pub fn stat_tree(dir: &Path) -> IoResult<StatMap> { ... }
```

Threads

```
pub fn stat_tree(dir: &Path) -> IoResult<StatMap> {
    let mut map = BTreeMap::new();

    let pool = TaskPool::new(10);
    let (tx, rx) = channel();
    let mut pending = 0us;

    try!(spawn_dir_stat(dir, &pool, tx.clone()));
    pending += 1;

    while pending > 0 { ... }

    Ok(map)
}
```

Threads

```
while pending > 0 {
    match rx.recv().unwrap() {
        Ok(stats) => {
            for (path, stat) in stats.into_iter() {
                if stat.kind == FileType::Directory {
                    try!(spawn_dir_stat(&path, &pool, tx.clone()));
                    pending += 1;
                }
                map.insert(path, stat);
            }
        },
        Err(e) => return Err(e)
    }
    pending -= 1;
}
```

```
fn spawn_dir_stat(dir: &Path, pool: &TaskPool,
                  tx: Sender<IoResult<Vec<(Path, FileStat)>>>)
    -> IoResult<()>
{
    let entries = try!(readdir(dir));
    pool.execute(move || {
        let mut stats = Vec::new();
        for path in entries.into_iter() {
            match path.lstat() {
                Ok(stat) => stats.push((path, stat)),
                Err(e) => {
                    tx.send(Err(e)).unwrap();
                    return;
                }
            }
        }
        tx.send(Ok(stats)).unwrap();
    });

    Ok(())
}
```

Threads

```
extern crate faststat;

fn main() {
    for arg in std::os::args().into_iter().skip(1) {
        match faststat::stat_tree(&Path::new(arg)) {
            Err(e) => {
                println!("{}", e);
                std::os::set_exit_status(1);
            },
            Ok(map) => {
                for (path, stat) in map.iter() {
                    println!("{:?}: {:?}", path, stat.modified);
                }
            }
        }
    }
}
```

The End

```
enum IntOrString {
    I(isize), S(String)
}

#[test]
fn corrupt_enum() {
    let mut s = IntOrString::S(String::new());
    match s {
        IntOrString::I(_) => (),
        IntOrString::S(ref p) => {
            s = IntOrString::I(0xdeadbeefis);
            // Now p is a &String, pointing at memory
            // that is an int of our choosing!
        }
    }
}
```

Types

Rust	C++

`enum Expr {`
`  Constant(i32),`
`  Negate(Box<Expr>),`
`  Add(Box<Expr>, Box<Expr>)`
`}`
	nothing

Box??

Types

```
fn eval(expr: &Box<Expr>) -> i32 {
    match expr {
        &box Expr::Constant(k)
            => k,
        &box Expr::Add(ref left, ref right)
            => eval(left) + eval(right),
        &box Expr::Negate(ref op)
            => -eval(op)
    }
}
```