Rust

cs378

Chris Rossbach

Outline

- Administrivia
  - Midterm 1 discussion
- Technical Agenda
  - Rust!
    - Overview
    - Decoupling Shared, Mutable, and State
    - Channels and Synchronization
  - Rust Lab Preview

Acknowledgements:

- https://www.slideshare.net/nikomatsakis/rust-concurrency-tutorial-2015-1202
- Thanks Nikolas Matsakis!

Rust Motivation

Rust Motivation

Locks’ litany of problems:

Rust Motivation

Locks’ litany of problems:

- Deadlock

Rust Motivation

Locks’ litany of problems:

- Deadlock
- Priority inversion

Rust Motivation

Locks’ litany of problems:

- Deadlock
- Priority inversion
- Convoys

Rust Motivation

Locks’ litany of problems:

- Deadlock
- Priority inversion
- Convoys
- Fault Isolation

Rust Motivation

Locks’ litany of problems:

- Deadlock
- Priority inversion
- Convoys
- Fault Isolation
- Preemption Tolerance

Rust Motivation

Locks’ litany of problems:

- Deadlock
- Priority inversion
- Convoys
- Fault Isolation
- Preemption Tolerance
- Performance

Rust Motivation

Locks’ litany of problems:

- Deadlock
- Priority inversion
- Convoys
- Fault Isolation
- Preemption Tolerance
- Performance
- Poor composability…

Rust Motivation

Locks' litany of problems:

- Deadlock
- Priority inversion
- Convoys
- Fault Isolation
- Preemption Tolerance
- Performance
- Poor composability…

Solution: don't use locks

- non-blocking
- Data-structure-centric
- HTM
- blah, blah, blah..

Rust Motivation

Locks' litany of problems:

- Deadlock
- Priority inversion
- Convoys
- Fault Isolation
- Preemption Tolerance
- Performance
- Poor composability…

Solu on't use lo

- on-b ng
- Data-struct entric
- M
- bla

Rust Motivation

Locks' litany of problems:

- Deadlock
- Priority inversion
- Convoys
- Fault Isolation
- Preemption Toleran
- Performance
- Poor composability.

Solu on't use lo

- n-b ng
- ata stru entric

Shared mutable state requires locks

- So…separate sharing and mutability
- Use type system to make concurrency safe
  - Ownership
  - Immutability
  - Careful library support for sync primitives

Rust Goals

- Multi-paradigm language modeled after C and C++
  - Functional, Imperative, Object-Oriented
- Primary Goals:
  - Safe Memory Management
  - Safe Concurrency and Concurrent Controls

Rust Goals

Multi-paradigm language modeled after C and C++

- Functional, Imperative, Object-Oriented

Primary Goals:

- Safe Memory Management
- Safe Concurrency and Concurrent Controls

Be Fast: systems programming
Be Safe: don’t crash

Memory Management

Memory Management

Rust: a “safe” environment for memory

- No Null, Dangling, or Wild Pointers

Memory Management

Rust: a “safe” environment for memory

- No Null, Dangling, or Wild Pointers

Objects are *immutable* by default

- User has more explicit control over mutability

Memory Management

Rust: a “safe” environment for memory

- No Null, Dangling, or Wild Pointers

Objects are *immutable* by default

- User has more explicit control over mutability

Declared variables must be initialized prior to execution

- A bit of a pain for static/global state

Unsafe

Credit: http://www.skiingforever.com/ski-tricks/

Unsafe

Functions determined unsafe via specific behavior

- Deference null or raw pointers
- Data Races
- Type Inheritance

Credit: http://www.skiingforever.com/ski-tricks/

Unsafe

Functions determined unsafe via specific behavior

- Deference null or raw pointers
- Data Races
- Type Inheritance

Using “unsafe” keyword → bypass compiler enforcement

- Don’t do it. Not for the lab, anyway

Credit: http://www.skiingforever.com/ski-tricks/

Unsafe

Functions determined unsafe via specific behavior

- Deference null or raw pointers
- Data Races
- Type Inheritance

Using “unsafe” keyword → bypass compiler enforcement

- Don’t do it. Not for the lab, anyway

The user deals with the integrity of the code

Credit: http://www.skiingforever.com/ski-tricks/

Other Relevant Features

- First-Class Functions and Closures
  - Similar to Lua, Go, …
- Algebraic data types (enums)
- Class Traits
  - Similar to Java interfaces
  - Allows classes to share aspects

Other Relevant Features

- First-Class Functions and Closures
  - Similar to Lua, Go, …
- Algebraic data types (enums)
- Class Traits
  - Similar to Java interfaces
  - Allows classes to share aspects

Hard to use/learn without awareness of these issues

Concurrency

Concurrency

Tasks → Rust’s threads

Concurrency

Tasks → Rust’s threads

Each task → stack and a heap

- Stack Memory Allocation – A Slot
- Heap Memory Allocation – A Box

Concurrency

Tasks → Rust’s threads

Each task → stack and a heap

- Stack Memory Allocation – A Slot
- Heap Memory Allocation – A Box

Tasks can share stack (portions) with other tasks

- These objects must be immutable

Concurrency

Tasks → Rust's threads

Each task → stack and a heap

- Stack Memory Allocation – A Slot
- Heap Memory Allocation – A Box

Tasks can share stack (portions) with other tasks

- These objects must be immutable

Task States: Running, Blocked, Failing, Dead

- Failing task: interrupted by another process
- Dead task: only viewable by other tasks

Concurrency

Tasks → Rust’s threads

Each task → stack and a heap

- Stack Memory Allocation – A Slot
- Heap Memory Allocation – A Box

Tasks can share stack (portions) with other tasks

- These objects must be immutable

Task States: Running, Blocked, Failing, Dead

- Failing task: interrupted by another process
- Dead task: only viewable by other tasks

Scheduling

- Each task → finite time-slice
- If task doesn’t finish, deferred until later
- “M:N scheduler”

Hello World

```
fn main() {
    println!("Hello, world!")
}
```

Ownership

Ownership

Ownership

n. The act, state, or right of possessing something

Ownership

Ownership

n. The act, state, or right of possessing something

Borrow

v. To receive something with the promise of returning it

Ownership

Ownership

n. The act, state, or right of possessing something

Borrow

v. To receive something with the promise of returning it

Ownership/Borrowing →

No need for a runtime

Memory safety (GC)

Data-race freedom

Ownership

Ownership

n. The act, state, or right of possessing something

Borrow

v. To receive something with the promise of returning it

Ownership/Borrowing →

No need for a runtime

Memory safety (GC)

Data-race freedom

MM Options:

- Managed languages: GC
- Native languages: manual management
- Rust: 3rd option: ***track ownership***

Ownership

Ownership

n. The act, state, or right of possessing something

Borrow

v. To receive something with the promise of returning it

Ownership/Borrowing →

No need for a runtime

Memory safety (GC)

Data-race freedom

MM Options:

- Managed languages: GC
- Native languages: manual management
- Rust: 3rd option: ***track ownership***

- Each value in Rust has a variable called its *owner*.
- There can only be one owner at a time.
- Owner goes out of scope→value will be dropped.

Ownership/Borrowing

```
fn main() {
  let name = format!("...");
  helper(name);
}
```

Ownership/Borrowing

```
fn main() {
  let name = format!("...");
  helper(name);
}
```

Ownership/Borrowing

```
fn main() {
  let name = format!("...");
  helper(name);
}
```

```
fn helper(name: String) {
  println!(“{}”, name);
}
```

Ownership/Borrowing

```
fn main() {
  let name = format!("...");
  helper(name);
  helper(name);
}
```

```
fn helper(name: String) {
  println!(“{}”, name);
}
```

Ownership/Borrowing

```
fn main() {
  let name = format!("...");
  helper(name);
  helper(name);
}
```

Error: use of moved value: `name`

```
fn helper(name: String) {
  println!(“{}”, name);
}
```

Ownership/Borrowing

```
fn main() {
  let name = format!("...");
  helper(name);
  helper(name);
}
```

Error: use of moved value: `name`

```
fn helper(name: String) {
  println!(“{}”, name);
}
```

Take ownership of a String

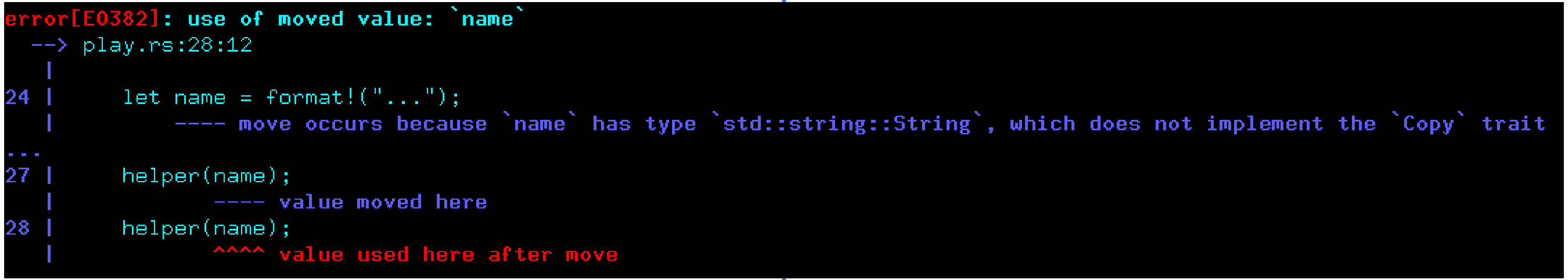
Ownership/Borrowing

```
fn main() {
  let name = format!("...");
  helper(name);
  helper(name);
}
```

Error: use of moved value: `name`

```
fn helper(name: String) {
  println!(“{}”, name);
}
```

Take ownership of a String



```
error[E0382]: use of moved value: `name`
  --> play.rs:28:12
   |
24 |     let name = format!("...");
   |         ---- move occurs because `name` has type `std::string::String`, which does not implement the `Copy` trait
...
27 |     helper(name);
   |            ---- value moved here
28 |     helper(name);
   |            ^^^^ value used here after move
```

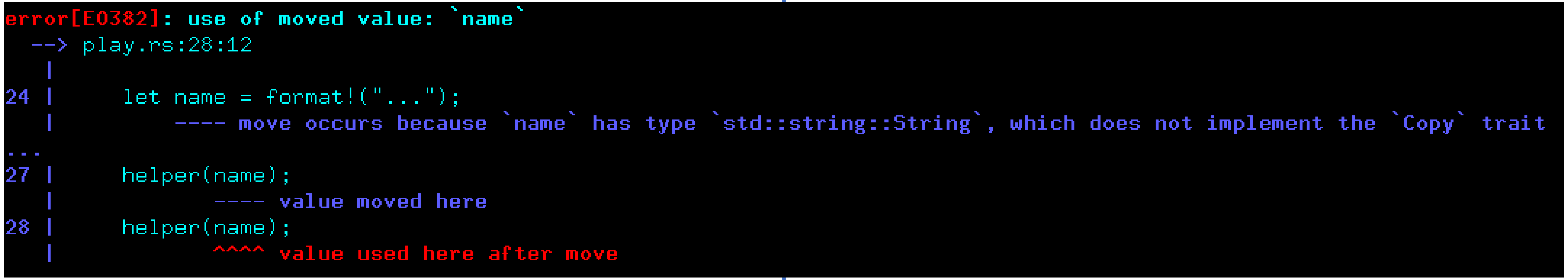
Ownership/Borrowing

```
fn main() {
  let name = format!("...");
  helper(name);
  helper(name);
}
```

Error: use of moved value: `name`

```
fn helper(name: String) {
  println!(“{}”, name);
}
```

Take ownership of a String



```
error[E0382]: use of moved value: `name`
  --> play.rs:28:12
   |
24 |     let name = format!("...");
   |         ---- move occurs because `name` has type `std::string::String`, which does not implement the `Copy` trait
...
27 |     helper(name);
   |            ---- value moved here
28 |     helper(name);
   |            ^^^^ value used here after move
```

What kinds of problems might this prevent?

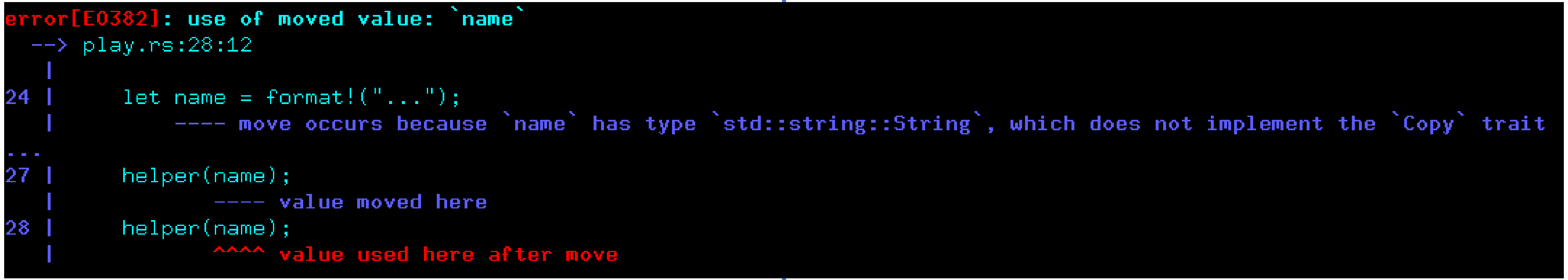
Ownership/Borrowing

```
fn main() {
  let name = format!("...");
  helper(name);
  helper(name);
}
```

Error: use of moved value: `name`

```
fn helper(name: String) {
  println!(“{}”, name);
}
```

Take ownership of a String



```
error[E0382]: use of moved value: `name`
  --> play.rs:28:12
   |
24 |     let name = format!("...");
   |         ---- move occurs because `name` has type `std::string::String`, which does not implement the `Copy` trait
...
27 |     helper(name);
   |            ---- value moved here
28 |     helper(name);
   |            ^^^^ value used here after move
```

What kinds of problems might this prevent?

Pass without ‘&’ takes “ownership implicitly” in other languages like Java

Shared Borrowing

```
fn main() {
  let name = format!("...");
  helper(&name);
  helper(&name);
}
```

```
fn helper(name: &String) {
  println!(“{}”, name);
}
```

Shared Borrowing

```
fn main() {
  let name = format!("...");
  helper(&name);
  helper(&name);
}
```

Lend the string

```
fn helper(name: &String) {
  println!(“{}”, name);
}
```

Shared Borrowing

```
fn main() {
  let name = format!("...");
  helper(&name);
  helper(&name);
}
```

Lend the string

```
fn helper(name: &String) {
  println!(“{}”, name);
}
```

Take a reference to a String

Shared Borrowing

```
fn main() {
  let name = format!("...");
  helper(&name);
  helper(&name);
}
```

Lend the string

```
fn helper(name: &String) {
  println!(“{}”, name);
}
```

Take a reference to a String

Why does this fix the problem?

Shared Borrowing with Concurrency

```
fn main() {
  let name = format!("...");
  helper(&name);
  helper(&name);
}
```

```
fn helper(name: &String) {
  thread::spawn(||{
    println!("{}", name);
  });
}
```

Shared Borrowing with Concurrency

```
fn main() {
  let name = format!("...");
  helper(&name);
  helper(&name);
}
```

```
fn helper(name: &String) {
  thread::spawn(||{
    println!("{}", name);
  });
}
```

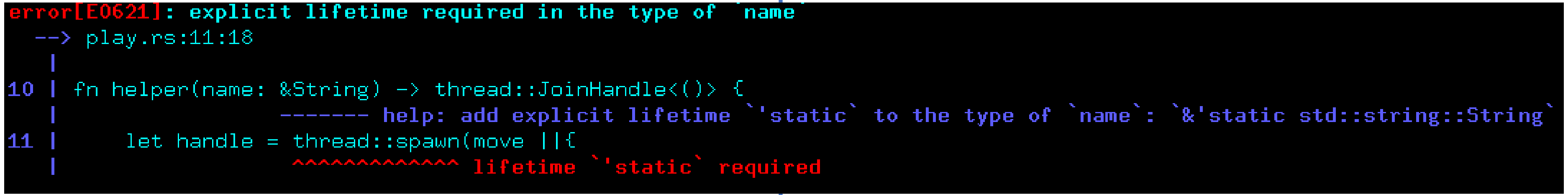
Lifetime `static` required

Shared Borrowing with Concurrency

```
fn main() {
  let name = format!("...");
  helper(&name);
  helper(&name);
}
```

```
fn helper(name: &String) {
  thread::spawn(||{
    println!("{}", name);
  });
}
```

Lifetime `static` required



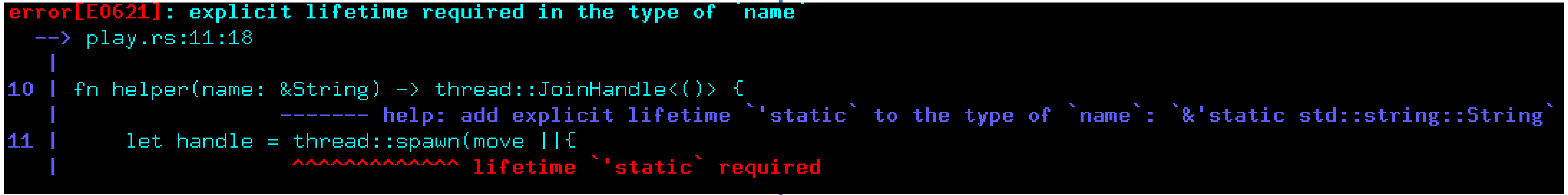
```
error[E0621]: explicit lifetime required in the type of `name`
  --> play.rs:11:18
   |
10 | fn helper(name: &String) -> thread::JoinHandle<()> {
   |                 ------- help: add explicit lifetime `'static` to the type of `name`: `&'static std::string::String`
11 |     let handle = thread::spawn(move ||{
   |                  ^^^^^^^^^^^^^ lifetime `'static` required
```

Shared Borrowing with Concurrency

```
fn main() {
  let name = format!("...");
  helper(&name);
  helper(&name);
}
```

```
fn helper(name: &String) {
  thread::spawn(||{
    println!("{}", name);
  });
}
```

Lifetime `static` required



```
error[E0621]: explicit lifetime required in the type of `name`
  --> play.rs:11:18
   |
10 | fn helper(name: &String) -> thread::JoinHandle<()> {
   |                 ------- help: add explicit lifetime `'static` to the type of `name`: `&'static std::string::String`
11 |     let handle = thread::spawn(move ||{
   |                  ^^^^^^^^^^^^^ lifetime `'static` required
```

Does this prevent the exact same class of problems?

Clone, Move

```rust
fn main() {
  let name = format!("...");
  helper(name.clone());
  helper(name);
}
```

```rust
fn helper(name: String) {
  thread::spawn(move || {
    println!("{}", name);
  });
}
```

Clone, Move

```
fn main() {
  let name = format!("...");
  helper(name.clone());
  helper(name);
}
```

```
fn helper(name: String) {
  thread::spawn(move || {
    println!("{}", name);
  });
}
```

Explicitly take ownership

Clone, Move

```
fn main() {
  let name = format!("...");
  helper(name.clone());
  helper(name);
}
```

Ensure concurrent owners
Work with different copies

```
fn helper(name: String) {
  thread::spawn(move || {
    println!("{}", name);
  });
}
```

Explicitly take ownership

Clone, Move

```
fn main() {
  let name = format!("...");
  helper(name.clone());
  helper(name);
}
```

Ensure concurrent owners
Work with different copies

Is this better?

```
fn helper(name: String) {
  thread::spawn(move || {
    println!("{}", name);
  });
}
```

Explicitly take ownership

Clone, Move

```
fn main() {
  let name = format!("...");
  helper(name.clone());
  helper(name);
}
```

Ensure concurrent owners
Work with different copies

Is this better?

```
fn helper(name: String) {
  thread::spawn(move || {
    println!("{}", name);
  });
}
```

Copy versus Clone:

Default: Types cannot be copied

- Values move from place to place
- E.g. file descriptor

Clone: Type is expensive to copy

- Make it explicit with clone call
- e.g. Hashtable

Copy: type implicitly copy-able

- e.g. u32, i32, f32, …

```
#[derive(Clone, Debug)]
```

Mutability

```
struct Structure {
    id: i32,
    map: HashMap<String, f32>,
}

impl Structure {
    fn mutate(&self, name: String, value: f32) {
        self.map.insert(name, value);
    }
}
```

Mutability

```
struct Structure {
    id: i32,
    map: HashMap<String, f32>,
}

impl Structure {
    fn mutate(&self, name: String, value: f32) {
        self.map.insert(name, value);
    }
}
```

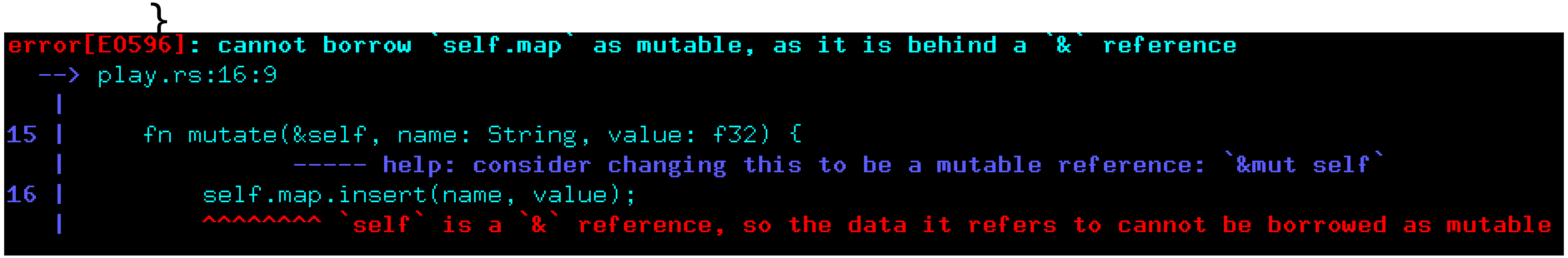
Error: cannot be borrowed as mutable

Mutability

```
struct Structure {
    id: i32,
    map: HashMap<String, f32>,
}

impl Structure {
    fn mutate(&self, name: String, value: f32) {
        self.map.insert(name, value);
    }
}
```

Error: cannot be borrowed as mutable



```
error[E0596]: cannot borrow `self.map` as mutable, as it is behind a `&` reference
  --> play.rs:16:9
   |
15 |     fn mutate(&self, name: String, value: f32) {
   |               ----- help: consider changing this to be a mutable reference: `&mut self`
16 |         self.map.insert(name, value);
   |         ^^^^^^^^ `self` is a `&` reference, so the data it refers to cannot be borrowed as mutable
```

Mutability

```
struct Structure {
    id: i32,
    map: HashMap<String, f32>,
}

impl Structure {
    fn mutate(&mut self, name: String, value: f32){
        self.map.insert(name, value);
    }
}
```

Mutability

```
struct Structure {
    id: i32,
    map: HashMap<String, f32>,
}

impl Structure {
    fn mutate(&mut self, name: String, value: f32){
        self.map.insert(name, value);
    }
}
```

Mutability

```
struct Structure {
    id: i32,
    map: HashMap<String, f32>,
}

impl Structure {
    fn mutate(&mut self, name: String, value: f32){
        self.map.insert(name, value);
    }
}
```

Key idea:

- Force mutation and ownership to be explicit
- Fixes MM *and* concurrency in fell swoop!

Sharing State: Channels

Sharing State: Channels

```
fn main() {
```

Sharing State: Channels

```
fn main() {
  let (tx0, rx0) = channel();
```

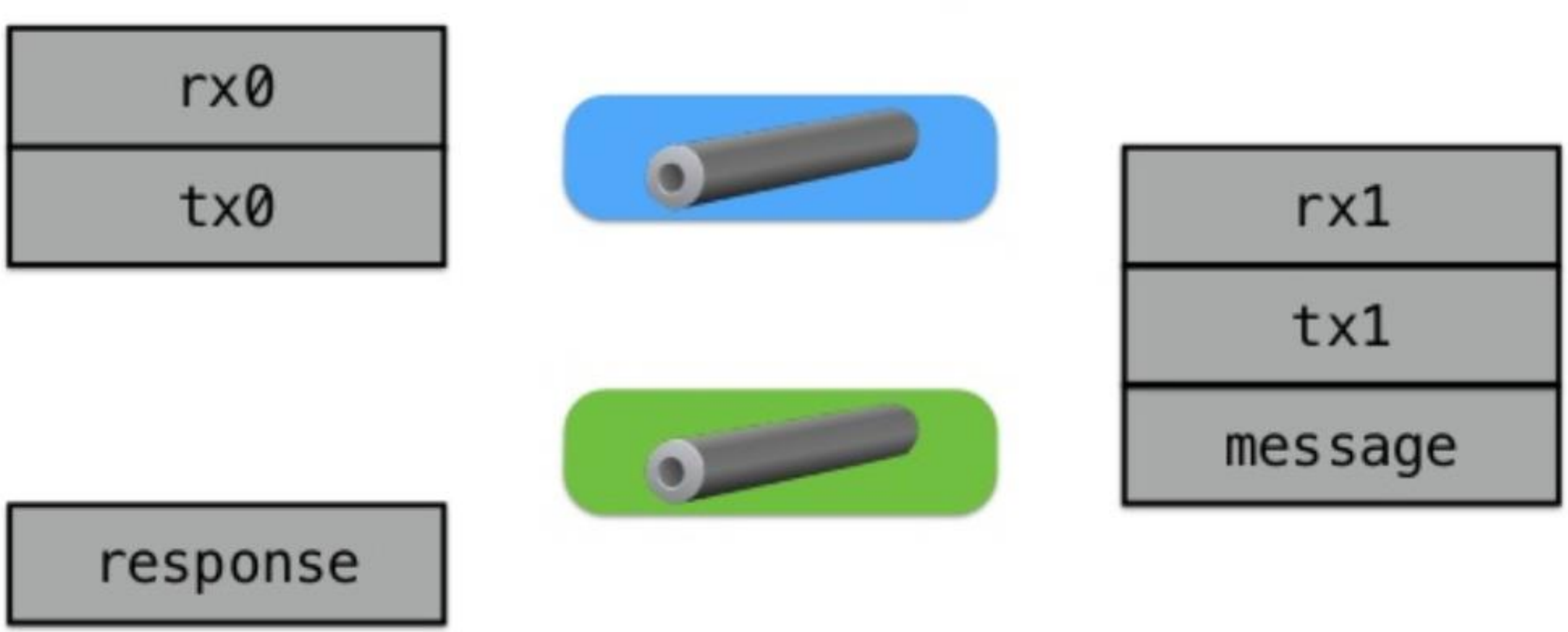
Sharing State: Channels

```
fn main() {
  let (tx0, rx0) = channel();
  thread::spawn(move || {
    let (tx1, rx1) = channel();
    tx0.send((format!("yo"), tx1)).unwrap();
    let response = rx1.recv().unwrap();
    println!("child got {}", response);
  });
```

Sharing State: Channels

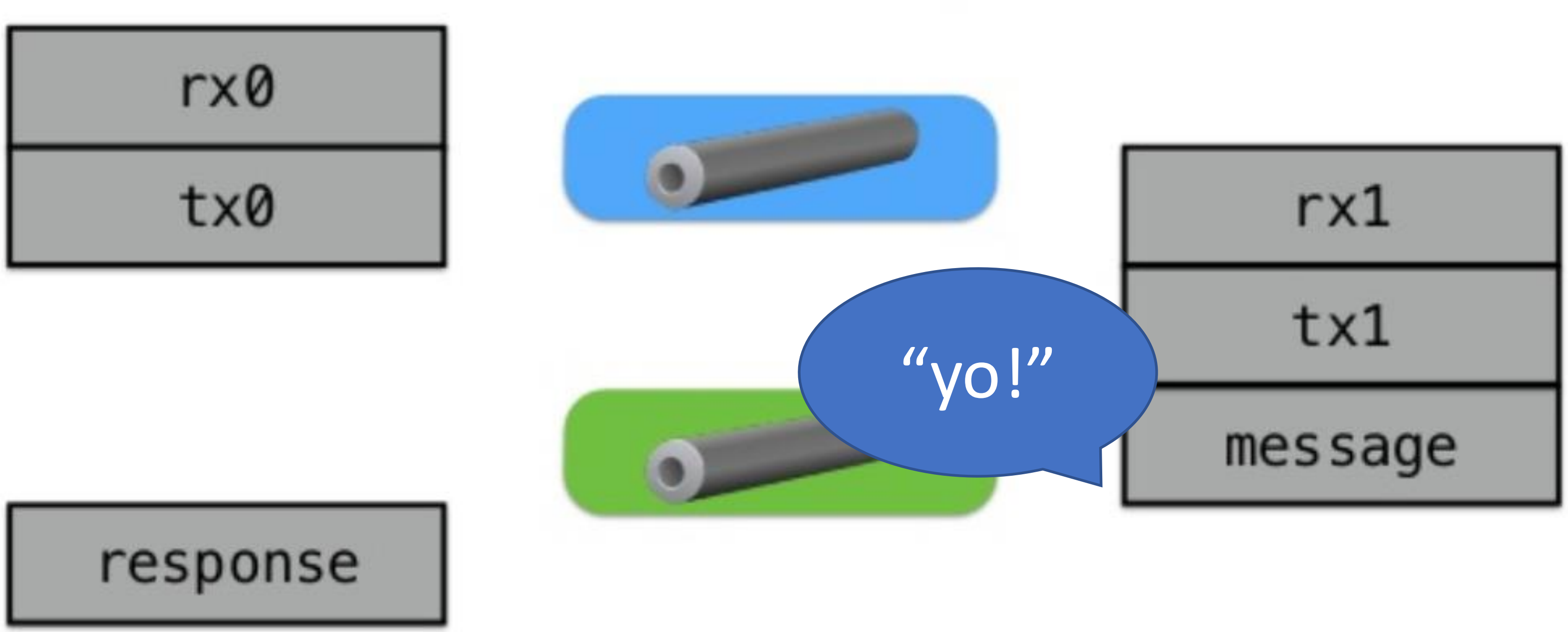
```rust
fn main() {
  let (tx0, rx0) = channel();
  thread::spawn(move || {
    let (tx1, rx1) = channel();
    tx0.send((format!("yo"), tx1)).unwrap();
    let response = rx1.recv().unwrap();
    println!("child got {}", response);
  });
  let (message, tx1) = rx0.recv().unwrap();
  tx1.send(format!("what up!")).unwrap();
  println("parent received {}", message);
}
```

Sharing State: Channels



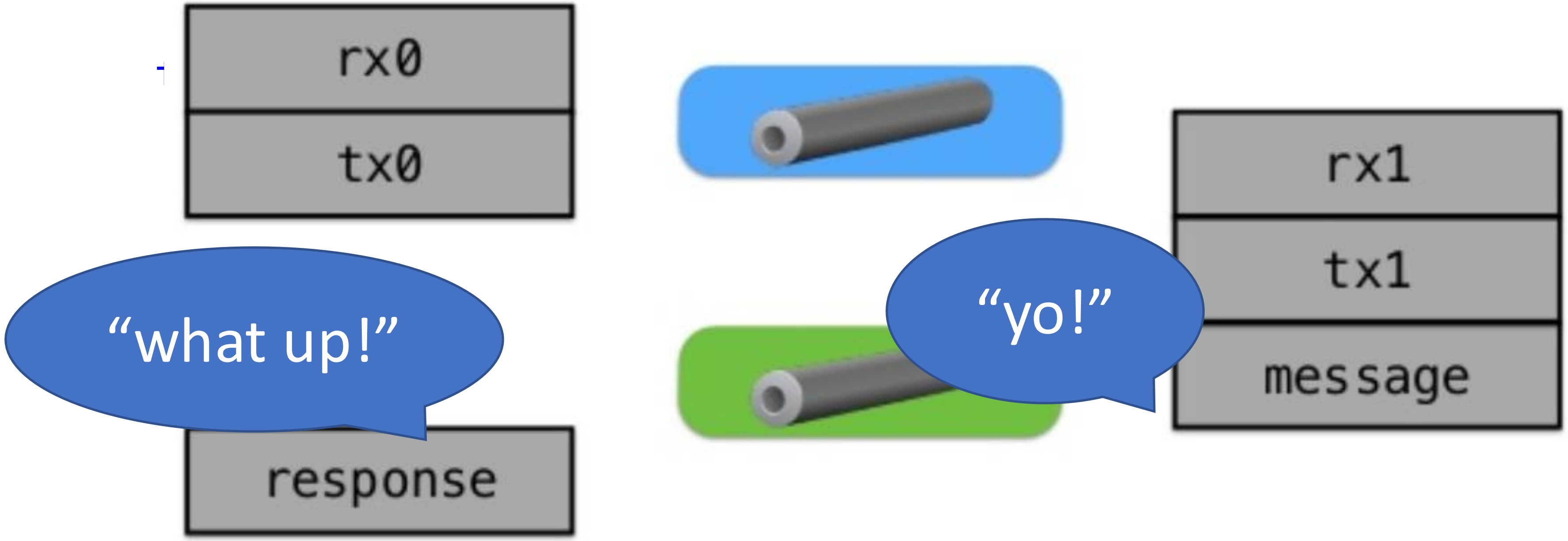
```
    let (message, tx1) = rx0.recv().unwrap();
    tx1.send(format!("what up!")).unwrap();
    println("parent received {}", message);
}
```

Sharing State: Channels



```
  let (message, tx1) = rx0.recv().unwrap();
  tx1.send(format!("what up!")).unwrap();
  println("parent received {}", message);
}
```

Sharing State: Channels



```
  let (message, tx1) = rx0.recv().unwrap();
  tx1.send(format!("what up!")).unwrap();
  println("parent received {}", message);
}
```

Sharing State: Channels

```
fn main() {
  let (tx0, rx0) = channel();
  thread::spawn(move || {
    let (tx1, rx1) = channel();
    tx0.send((format!("yo"), tx1)).unwrap();
    let response = rx1.recv().unwrap();
    println!("child got {}", response);
  });
  let (message, tx1) = rx0.recv().unwrap();
  tx1.send(format!("what up!")).unwrap();
  println("parent received {}", message);
}
```

Sharing State: Channels

```
fn main() {
  let (tx0, rx0) = channel();
  thread::spawn(move || {
    let (tx1, rx1) = channel();
    tx0.send((format!("yo"), tx1)).unwrap();
    let response = rx1.recv().unwrap();
    println!("child got {}", response);
  });
  let (message, tx1) = rx0.recv().unwrap();
  tx1.send(format!("what up!")).unwrap();
  println("parent received {}", message);
}
```

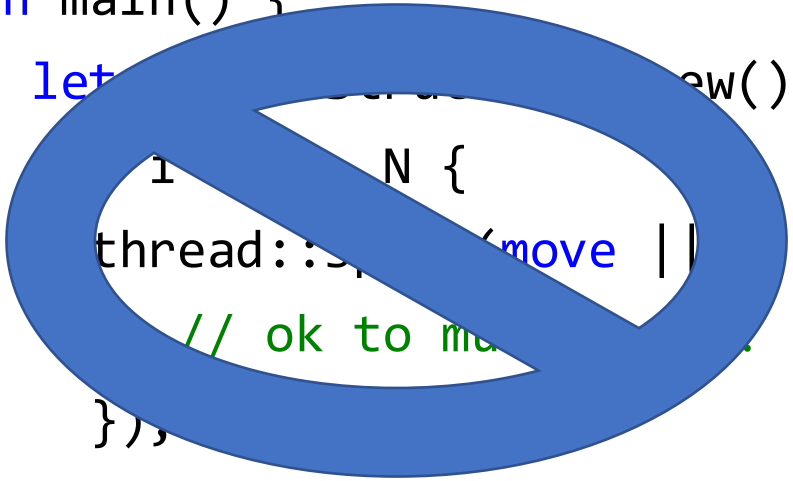
APIs return Option<T>

Sharing State

```
fn main() {
  let var = Structure::new();
  for i in 0..N {
    thread::spawn(move || {
       // ok to mutate var?
    });
  }
}
```

Sharing State

```
fn main() {
  let                     ew();
    i       N {
      thread::s        (move ||
        // ok to mu
      });
  }
}
```



Sharing State: Arc and Mutex

```
fn main() {
  let var = Structure::new();
  let var_lock = Mutex::new(var);
  let var_arc = Arc::new(var_lock);
  for i in 0..N {
    thread::spawn(move || {
      let ldata = Arc::clone(&var_arc);
      let vdata = ldata.lock();
      // ok to mutate var (vdata)!
    });
  }
}
```

Sharing State: Arc and Mutex

```
fn main() {
  let var = Structure::new();
  let var_lock = Mutex::new(var);
  let var_arc = Arc::new(var_lock);
  for i in 0..N {
    thread::spawn(move || {
      let ldata = Arc::clone(&var_arc);
      let vdata = ldata.lock();
      // ok to mutate var (vdata)!
    });
  }
}
```

Sharing State: Arc and Mutex

```
fn main() {
  let var = Structure::new();
  let var_lock = Mutex::new(var);
  let var_arc = Arc::new(var_lock);
  for i in 0..N {
    thread::spawn(move || {
      let ldata = Arc::clone(&var_arc);
      let vdata = ldata.lock();
      // ok to mutate var (vdata)!
    });
  }
}
```

Sharing State: Arc and Mutex

```
fn main() {
  let var = Structure::new();
  let var_lock = Mutex::new(var);
  let var_arc = Arc::new(var_lock);
  for i in 0..N {
    thread::spawn(move || {
      let ldata = Arc::clone(&var_arc);
      let vdata = ldata.lock();
      // ok to mutate var (vdata)!
    });
  }
}
```

Sharing State: Arc and Mutex

```
fn main() {
  let var = Structure::new();
  let var_lock = Mutex::new(var);
  let var_arc = Arc::new(var_lock);
  for i in 0..N {
    thread::spawn(move || {
      let ldata = Arc::clone(&var_arc);
      let vdata = ldata.lock();
      // ok to mutate var (vdata)!
    });
  }
}
```

Sharing State: Arc and Mutex

```
fn main() {
  let var = Structure::new();
  let var_lock = Mutex::new(var);
  let var_arc = Arc::new(var_lock);
  for i in 0..N {
    thread::spawn(move || {
      let ldata = Arc::clone(&var_arc);
      let vdata = ldata.lock();
      // ok to mutate var (vdata)!
    });
  }
}
```

Key ideas:

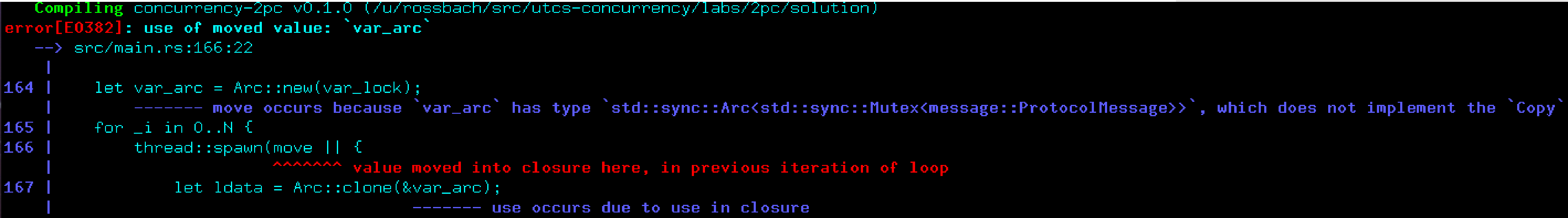
- Use reference counting wrapper to pass refs
- Use scoped lock for mutual exclusion
- Actually compiles → works 1st time!

Sharing State: Arc and Mutex, *really*

```
fn test() {
  let var = Structure::new();
  let var_lock = Mutex::new(var);
  let var_arc = Arc::new(var_lock);
  for i in 0..N {
    thread::spawn(move || {
      let ldata = Arc::clone(&var_arc);
      let vdata = ldata.lock();
      // ok to mutate var (vdata)!
    });
  }
}
```

Sharing State: Arc and Mutex, *really*

```
fn test() {
  let var = Structure::new();
  let var_lock = Mutex::new(var);
  let var_arc = Arc::new(var_lock);
  for i in 0..N {
    thread::spawn(move || {
      let ldata = Arc::clone(&var_arc);
      let vdata = ldata.lock();
      // ok to mutate var (vdata)!
    });
```

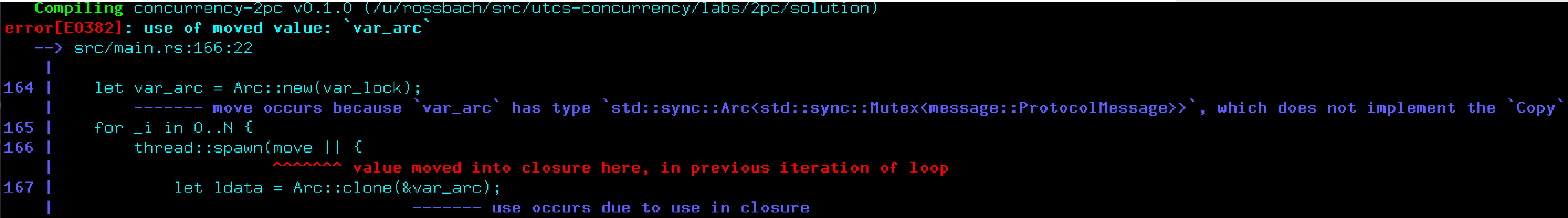


```
   Compiling concurrency-2pc v0.1.0 (/u/rossbach/src/utcs-concurrency/labs/2pc/solution)
error[E0382]: use of moved value: `var_arc`
   --> src/main.rs:166:22
    |
164 |     let var_arc = Arc::new(var_lock);
    |         ------- move occurs because `var_arc` has type `std::sync::Arc<std::sync::Mutex<message::ProtocolMessage>>`, which does not implement the `Copy`
165 |     for _i in 0..N {
166 |         thread::spawn(move || {
    |                       ^^^^^^^ value moved into closure here, in previous iteration of loop
167 |             let ldata = Arc::clone(&var_arc);
    |                                    ------- use occurs due to use in closure
```

Sharing State: Arc and Mutex, *really*

```
fn test() {
  let var = Structure::new();
  let var_lock = Mutex::new(var);
  let var_arc = Arc::new(var_lock);
  for i in 0..N {
    thread::spawn(move || {
      let ldata = Arc::clone(&var_arc);
      let vdata = ldata.lock();
      // ok to mutate var (vdata)!
    });
```

Why doesn’t “&” fix it?
(&var_arc, instead of just var_arc)



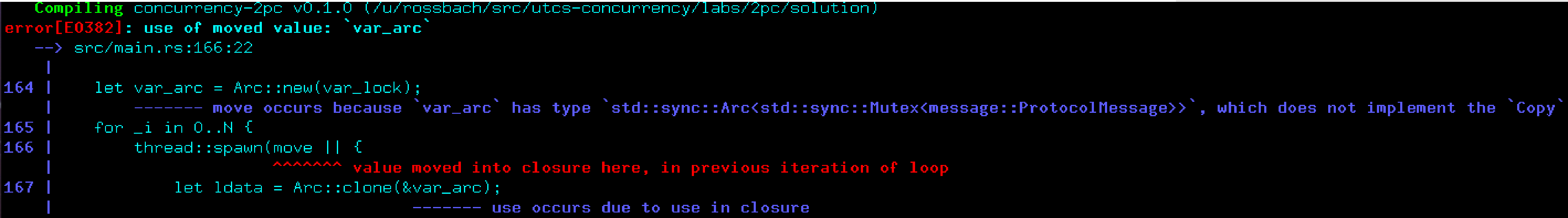
```
   Compiling concurrency-2pc v0.1.0 (/u/rossbach/src/utcs-concurrency/labs/2pc/solution)
error[E0382]: use of moved value: `var_arc`
   --> src/main.rs:166:22
    |
164 |     let var_arc = Arc::new(var_lock);
    |         ------- move occurs because `var_arc` has type `std::sync::Arc<std::sync::Mutex<message::ProtocolMessage>>`, which does not implement the `Copy`
165 |     for _i in 0..N {
166 |         thread::spawn(move || {
    |                       ^^^^^^^ value moved into closure here, in previous iteration of loop
167 |             let ldata = Arc::clone(&var_arc);
    |                                     ------- use occurs due to use in closure
```

Sharing State: Arc and Mutex, *really*

```
fn test() {
  let var = Structure::new();
  let var_lock = Mutex::new(var);
  let var_arc = Arc::new(var_lock);
  for i in 0..N {
    thread::spawn(move || {
      let ldata = Arc::clone(&var_arc);
      let vdata = ldata.lock();
      // ok to mutate var (vdata)!
    });
```

Why doesn’t “&” fix it?
(&var_arc, instead of just var_arc)

Would cloning var_arc fix it?



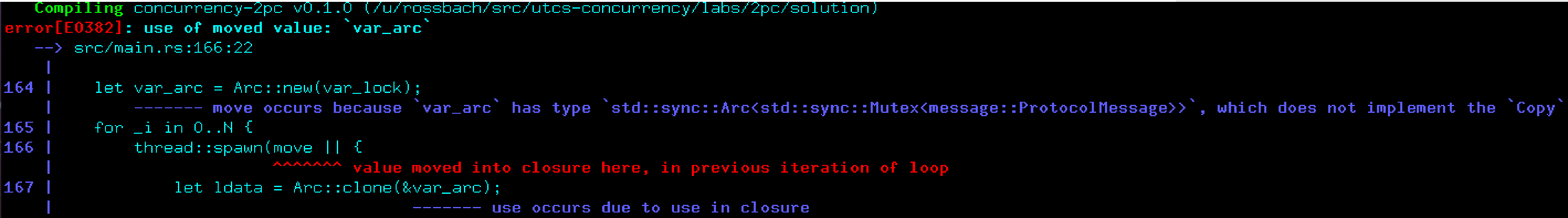
```
   Compiling concurrency-2pc v0.1.0 (/u/rossbach/src/utcs-concurrency/labs/2pc/solution)
error[E0382]: use of moved value: `var_arc`
   --> src/main.rs:166:22
    |
164 |     let var_arc = Arc::new(var_lock);
    |         ------- move occurs because `var_arc` has type `std::sync::Arc<std::sync::Mutex<message::ProtocolMessage>>`, which does not implement the `Copy`
165 |     for _i in 0..N {
166 |         thread::spawn(move || {
    |                       ^^^^^^^ value moved into closure here, in previous iteration of loop
167 |             let ldata = Arc::clone(&var_arc);
    |                                     ------- use occurs due to use in closure
```

Sharing State: Arc and Mutex, *really*

```
fn test() {
  let var = Structure::new();
  let var_lock = Mutex::new(var);
  let var_arc = Arc::new(var_lock);
  for i in 0..N {
    thread::spawn(move || {
      let ldata = Arc::clone(&var_arc.clone());
      let vdata = ldata.lock();
      // ok to mutate var (vdata)!
    });
  }
}
```

Sharing State: Arc and Mutex, *really*

```
fn test() {
  let var = Structure::new();
  let var_lock = Mutex::new(var);
  let var_arc = Arc::new(var_lock);
  for i in 0..N {
    thread::spawn(move || {
      let ldata = Arc::clone(&var_arc.clone());
      let vdata = ldata.lock();
      // ok to mutate var (vdata)!
    });
```

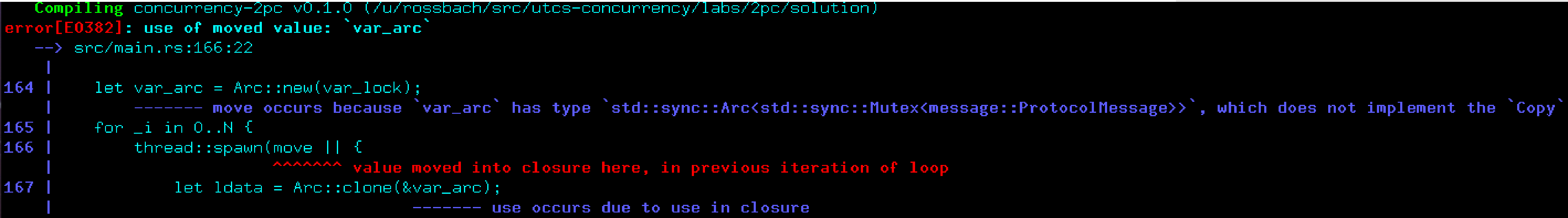


```
   Compiling concurrency-2pc v0.1.0 (/u/rossbach/src/utcs-concurrency/labs/2pc/solution)
error[E0382]: use of moved value: `var_arc`
   --> src/main.rs:166:22
    |
164 |     let var_arc = Arc::new(var_lock);
    |         ------- move occurs because `var_arc` has type `std::sync::Arc<std::sync::Mutex<message::ProtocolMessage>>`, which does not implement the `Copy`
165 |     for _i in 0..N {
166 |         thread::spawn(move || {
    |                       ^^^^^^^ value moved into closure here, in previous iteration of loop
167 |             let ldata = Arc::clone(&var_arc);
    |                                     ------- use occurs due to use in closure
```

Sharing State: Arc and Mutex, *really*

```
fn test() {
  let var = Structure::new();
  let var_lock = Mutex::new(var);
  let var_arc = Arc::new(var_lock);
  for i in 0..N {
    thread::spawn(move || {
      let ldata = Arc::clone(&var_arc.clone());
      let vdata = ldata.lock();
      // ok to mutate var (vdata)!
    });
```

Same problem!



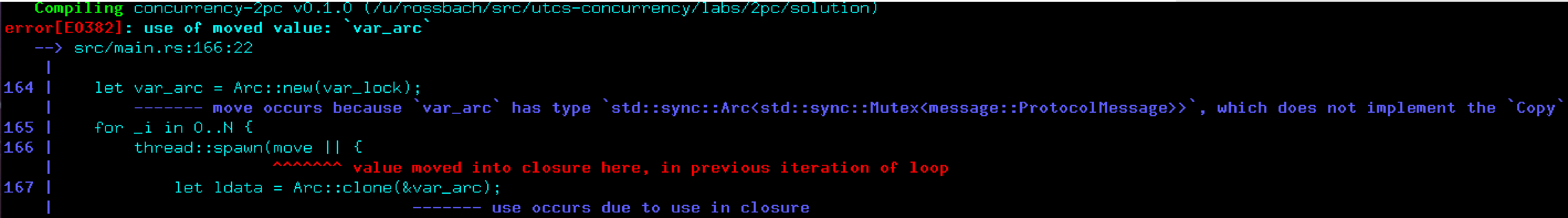
```
   Compiling concurrency-2pc v0.1.0 (/u/rossbach/src/utcs-concurrency/labs/2pc/solution)
error[E0382]: use of moved value: `var_arc`
   --> src/main.rs:166:22
    |
164 |     let var_arc = Arc::new(var_lock);
    |         ------- move occurs because `var_arc` has type `std::sync::Arc<std::sync::Mutex<message::ProtocolMessage>>`, which does not implement the `Copy`
165 |     for _i in 0..N {
166 |         thread::spawn(move || {
    |                       ^^^^^^^ value moved into closure here, in previous iteration of loop
167 |             let ldata = Arc::clone(&var_arc);
    |                                     ------- use occurs due to use in closure
```

Sharing State: Arc and Mutex, *really*

```
fn test() {
  let var = Structure::new();
  let var_lock = Mutex::new(var);
  let var_arc = Arc::new(var_lock);
  for i in 0..N {
    thread::spawn(move || {
      let ldata = Arc::clone(&var_arc.clone());
      let vdata = ldata.lock();
      // ok to mutate var (vdata)!
    });
```

Same problem!

What if we just don't ***move?***



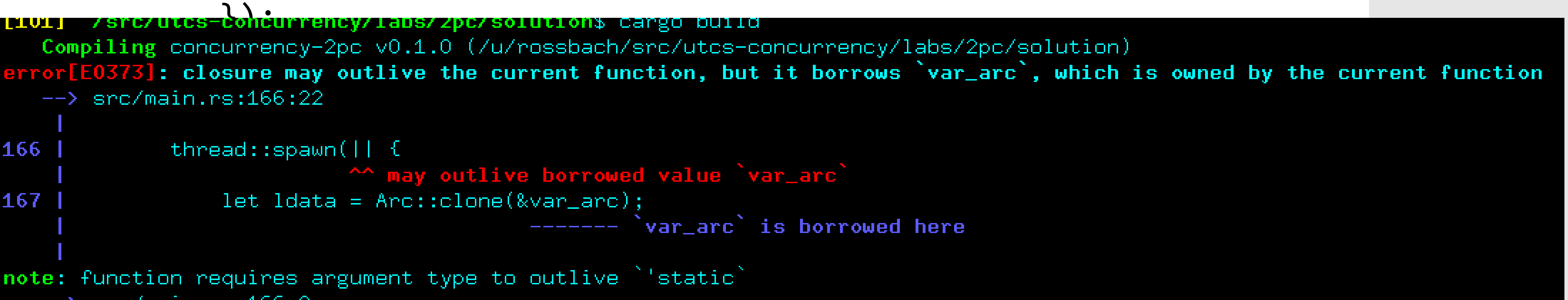
```
   Compiling concurrency-2pc v0.1.0 (/u/rossbach/src/utcs-concurrency/labs/2pc/solution)
error[E0382]: use of moved value: `var_arc`
   --> src/main.rs:166:22
    |
164 |     let var_arc = Arc::new(var_lock);
    |         ------- move occurs because `var_arc` has type `std::sync::Arc<std::sync::Mutex<message::ProtocolMessage>>`, which does not implement the `Copy`
165 |     for _i in 0..N {
166 |         thread::spawn(move || {
    |                       ^^^^^^^ value moved into closure here, in previous iteration of loop
167 |             let ldata = Arc::clone(&var_arc);
    |                                     ------- use occurs due to use in closure
```

Sharing State: Arc and Mutex, *really*

```
fn test() {
  let var = Structure::new();
  let var_lock = Mutex::new(var);
  let var_arc = Arc::new(var_lock);
  for i in 0..N {
    thread::spawn(|| {
      let ldata = Arc::clone(&var_arc);
      let vdata = ldata.lock();
      // ok to mutate var (vdata)!
    });
  }
}
```

Sharing State: Arc and Mutex, *really*

```
fn test() {
  let var = Structure::new();
  let var_lock = Mutex::new(var);
  let var_arc = Arc::new(var_lock);
  for i in 0..N {
    thread::spawn(|| {
      let ldata = Arc::clone(&var_arc);
      let vdata = ldata.lock();
      // ok to mutate var (vdata)!
    });
```

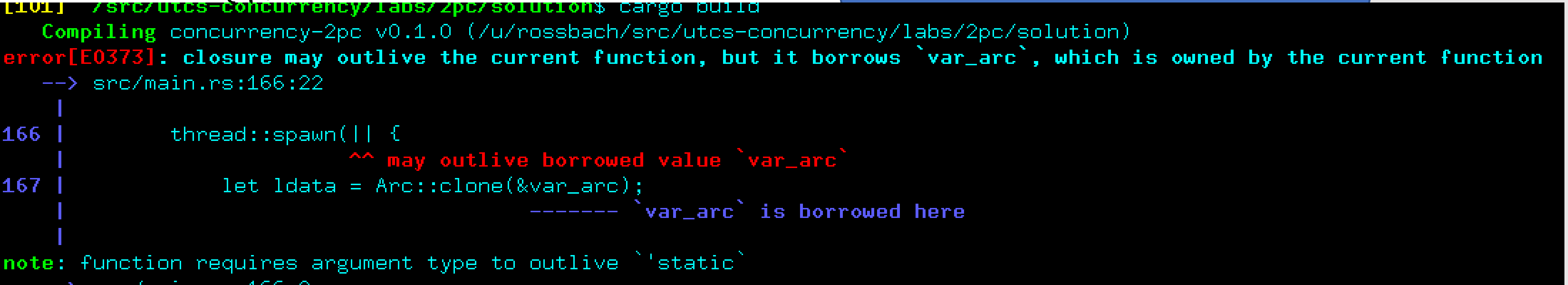


```
[101] ~/src/utcs-concurrency/labs/2pc/solution$ cargo build
   Compiling concurrency-2pc v0.1.0 (/u/rossbach/src/utcs-concurrency/labs/2pc/solution)
error[E0373]: closure may outlive the current function, but it borrows `var_arc`, which is owned by the current function
   --> src/main.rs:166:22
    |
166 |         thread::spawn(|| {
    |                       ^^ may outlive borrowed value `var_arc`
167 |             let ldata = Arc::clone(&var_arc);
    |                                     -------- `var_arc` is borrowed here
    |
note: function requires argument type to outlive `'static`
```

Sharing State: Arc and Mutex, *really*

```
fn test() {
  let var = Structure::new();
  let var_lock = Mutex::new(var);
  let var_arc = Arc::new(var_lock);
  for i in 0..N {
    thread::spawn(|| {
      let ldata = Arc::clone(&var_arc);
      let vdata = ldata.lock();
      // ok to mutate var (vdata)!
    });
```

What’s the actual fix?



```
[101] /src/utcs-concurrency/labs/2pc/solution$ cargo build
   Compiling concurrency-2pc v0.1.0 (/u/rossbach/src/utcs-concurrency/labs/2pc/solution)
error[E0373]: closure may outlive the current function, but it borrows `var_arc`, which is owned by the current function
   --> src/main.rs:166:22
    |
166 |         thread::spawn(|| {
    |                       ^^ may outlive borrowed value `var_arc`
167 |             let ldata = Arc::clone(&var_arc);
    |                                     ------- `var_arc` is borrowed here
    |
note: function requires argument type to outlive `'static`
```

Sharing State: Arc and Mutex, *really*

```
fn test() {
  let var = Structure::new();
  let var_lock = Mutex::new(var);
  let var_arc = Arc::new(var_lock);
  for i in 0..N {
    let clone_arc = var_arc.clone();
    thread::spawn(move || {
      let ldata = Arc::clone(&clone_arc);
      let vdata = ldata.lock();
      // ok to mutate var (vdata)!
    });
  }
}
```

Sharing State: Arc and Mutex, *really*

```
fn test() {
  let var = Structure::new();
  let var_lock = Mutex::new(var);
  let var_arc = Arc::new(var_lock);
  for i in 0..N {
    let clone_arc = var_arc.clone();
    thread::spawn(move || {
      let ldata = Arc::clone(&clone_arc);
      let vdata = ldata.lock();
      // ok to mutate var (vdata)!
    });
  }
}
```

Compiles! Yay!
Other fixes?

Sharing State: Arc and Mutex, *really*

```
fn test() {
  let var = Structure::new();
  let var_lock = Mutex::new(var);
  let var_arc = Arc::new(var_lock);
  for i in 0..N {
    thread::spawn(move || {
      let ldata = Arc::clone(&var_arc);
      let vdata = ldata.lock();
      // ok to mutate var (vdata)!
    });
  }

}
```

Sharing State: Arc and Mutex, *really*

```
fn test() {
  let var = Structure::new();
  let var_lock = Mutex::new(var);
  let var_arc = Arc::new(var_lock);
  ~~for i in 0..N {~~
    thread::spawn(move || {
      let ldata = Arc::clone(&var_arc);
      let vdata = ldata.lock();
      // ok to mutate var (vdata)!
    });
  ~~}~~

}
```

Sharing State: Arc and Mutex, *really*

```
fn test() {
  let var = Structure::new();
  let var_lock = Mutex::new(var);
  let var_arc = Arc::new(var_lock);
  ~~for i in 0..N {~~
    thread::spawn(move || {
      let ldata = Arc::clone(&var_arc);
      let vdata = ldata.lock();
      // ok to mutate var (vdata)!
    });
  ~~}~~

}
```

Why does this compile?

Sharing State: Arc and Mutex, *really*

```
fn test() {
  let var = Structure::new();
  let var_lock = Mutex::new(var);
  let var_arc = Arc::new(var_lock);
  for i in 0..N {
    thread::spawn(~~move~~ || {
      let ldata = Arc::clone(&var_arc);
      let vdata = ldata.lock();
      // ok to mutate var (vdata)!
    });
  }

}
```

Could we use a vec of JoinHandle to keep var_arc in scope?

Sharing State: Arc and Mutex, *really*

```
fn test() {
  let var = Structure::new();
  let var_lock = Mutex::new(var);
  let var_arc = Arc::new(var_lock);
  for i in 0..N {
    thread::spawn(~~move~~ || {
      let ldata = Arc::clone(&var_arc);
      let vdata = ldata.lock();
      // ok to mutate var (vdata)!
    });
  }
  for i in 0..N { join(); }
}
```

Could we use a vec of JoinHandle to keep var_arc in scope?

Sharing State: Arc and Mutex, *really*

```
fn test() {
  let var = Structure::new();
  let var_lock = Mutex::new(var);
  let var_arc = Arc::new(var_lock);
  for i in 0..N {
    thread::spawn(~~move~~ || {
      let ldata = Arc::clone(&var_arc);
      let vdata = ldata.lock();
      // ok to mutate var (vdata)!
    });
  }
  for i in 0..N { join(); }
}
```

Could we use a vec of JoinHandle to keep var_arc in scope?

What if I need my lambda to own some things and borrow others?

Sharing State: Arc and Mutex, *really*

```
fn test() {
  let var = Structure::new();
  let var_lock = Mutex::new(var);
  let var_arc = Arc::new(var_lock);
  for i in 0..N {
    thread::spawn(~~move~~ || {
      let ldata = Arc::clone(&var_arc);
      let vdata = ldata.lock();
      // ok to mutate var (vdata)!
    });
  }
  for i in 0..N { join(); }
}
```

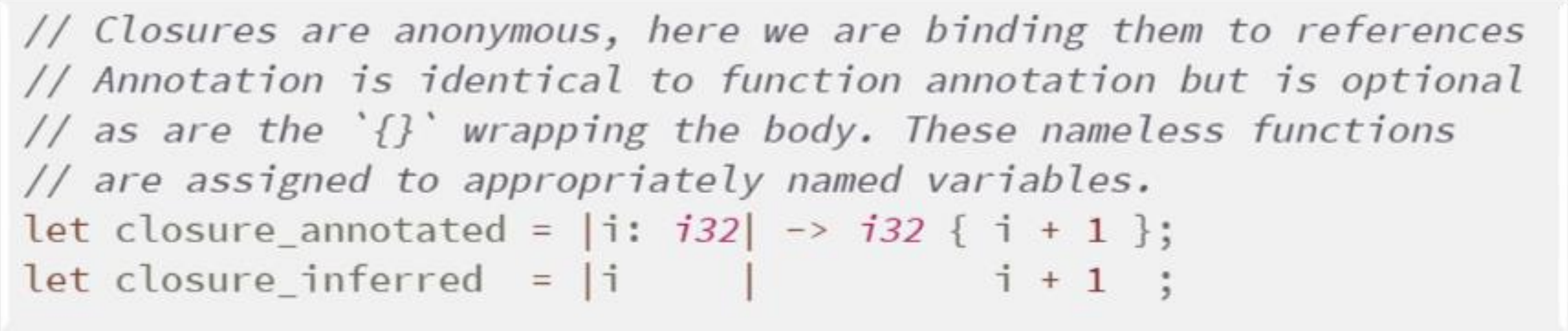
Parameters!

Could we use a vec of JoinHandle to keep var_arc in scope?

What if I need my lambda to own some things and borrow others?

Sharing State: Arc and Mutex, *really*

```
fn test() {
  let var = Structure::new();
  let var_lock = Mutex::new(var);
  let var_arc = Arc::new(var_lock);
```



```
// Closures are anonymous, here we are binding them to references
// Annotation is identical to function annotation but is optional
// as are the `{}` wrapping the body. These nameless functions
// are assigned to appropriately named variables.
let closure_annotated = |i: i32| -> i32 { i + 1 };
let closure_inferred  = |i     |          i + 1  ;
```

```
        // ok to mutate var (vdata)!
      });
  }
  for i in 0..N { join(); }
}
```

Parameters!

Could we use a vec of JoinHandle to keep var_arc in scope?

What if I need my lambda to own some things and borrow others?

Discussion

GC lambdas, Rust C++

- This is pretty nuanced:
- Stack closures, owned closures, managed closures, exchg heaps

Ownership and Macros

Macros use regexp and expand to closures

Summary

Rust: best of both worlds

systems vs productivity language

Separate sharing, mutability, concurrency

Type safety solves MM and concurrency

Have fun with the lab!