



Rust Wrapup

cs378h

Chris Rossbach



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

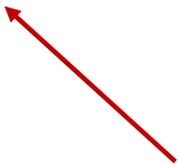
```
fn main() {  
    let name = format!(". . .");  
    helper(name);  
    helper(name);  
}
```

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

Ownership/Borrowing

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

Error: use of moved value: `name`

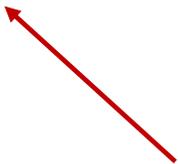


```
fn helper(name: String) {  
    println!("{}", name);  
}
```

Ownership/Borrowing

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

Error: use of moved value: `name`



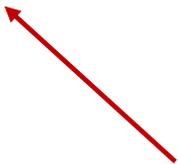
```
fn helper(name: String) {  
    println!("{}", name);  
}
```

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);  
}
```

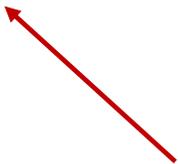


Take ownership of a String

```
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 by reference 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

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

```
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(|| {  
        println!("{}", name);  
    });  
}
```

Explicitly take ownership

Clone, Move

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

Ensure concurrent owners
Work with different copies

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

Explicitly take ownership

Clone, Move

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

Ensure concurrent owners
Work with different copies

Is this better?

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

Explicitly take ownership

Clone, Move

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

Ensure concurrent owners
Work with different copies

Is this better?

```
fn helper(name: String) {  
    thread::spawn(|| {  
        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

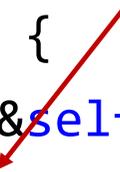
```
struct Structure {  
    id: i32,  
    map: HashMap<String, f32>,  
}  
  
impl Structure {  
    fn mutate(&self, name: String, value: f32) {  
        self.map.insert(name, value);  
    }  
}
```

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

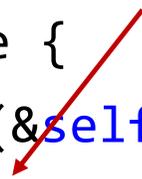


Mutability

```
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( name: String, value: f32){  
        self.map.insert(name, value);  
    }  
}
```

Mutability

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

```
impl Structure {  
    fn mutate( 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();
```

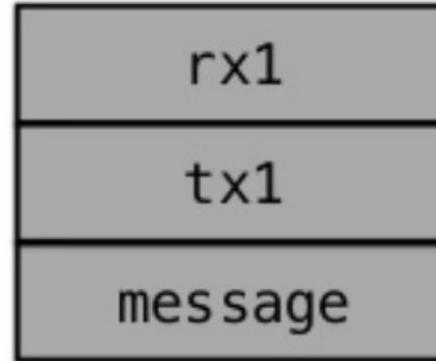
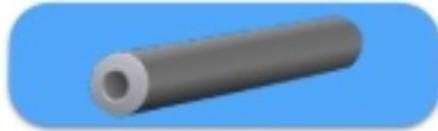
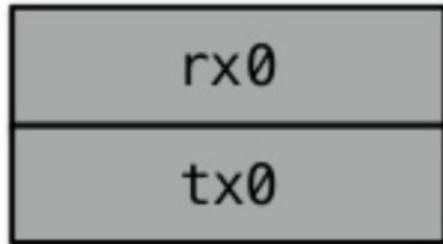
Sharing State: Channels

```
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

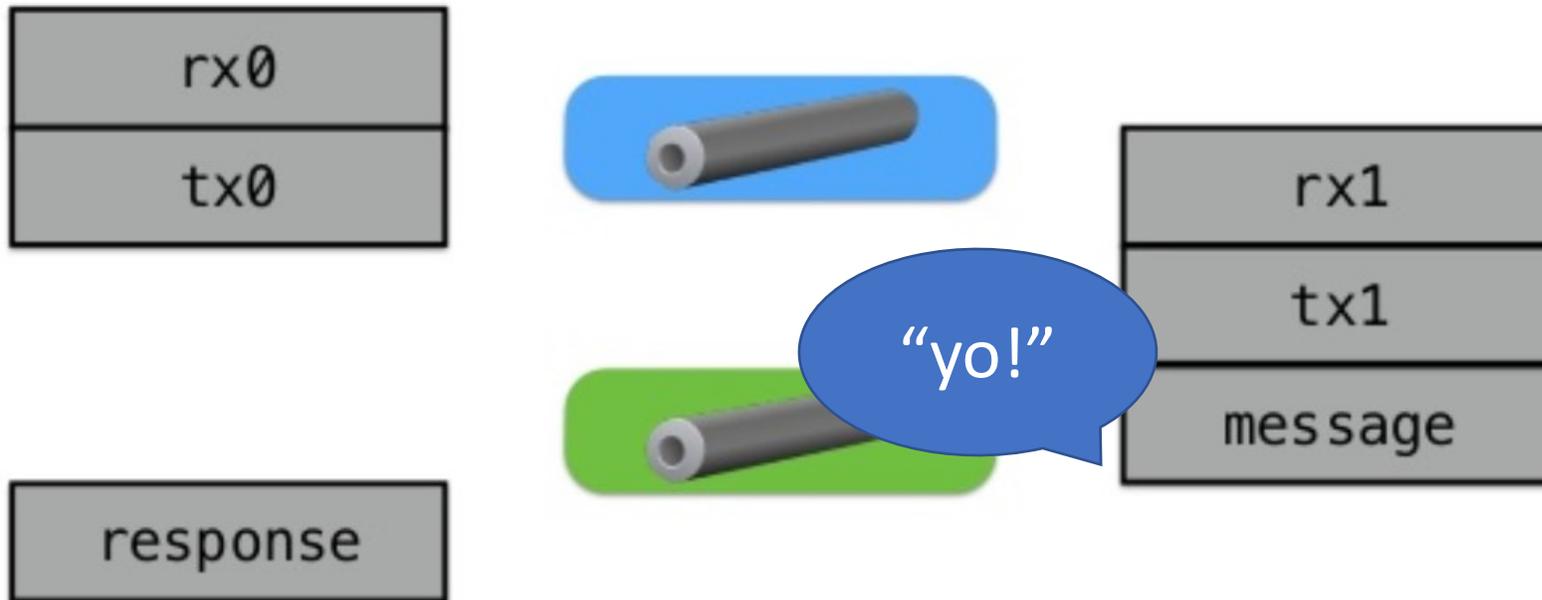
```
fn main() {  
    let (tx0, rx0) = channel();  
    thread::spawn(move || {  
        let (tx1, rx1) = channel();  
        tx0.send((format!("yo"), tx1)).unwrap();  
        let response = rx1.recv().unwrap();  
        println!("child got {}", response);  
    });  
    let (message, tx1) = rx0.recv().unwrap();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

Sharing State: Channels



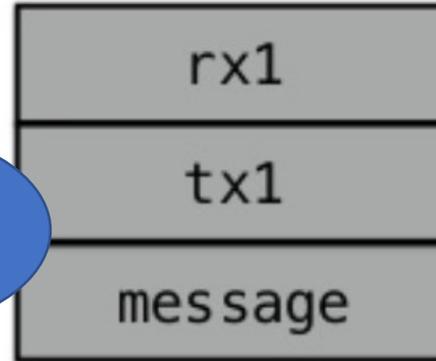
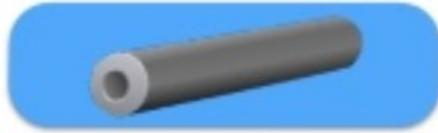
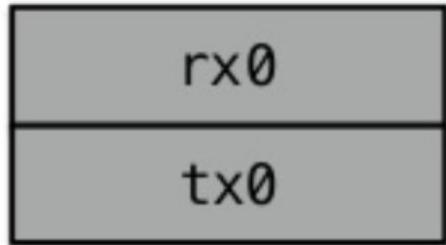
```
let (message, tx1) = rx0.recv().unwrap();  
tx1.send(format!("what up!")).unwrap();  
println("parent received {}", message);  
}
```

Sharing State: Channels



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

Sharing State: Channels



“what up!”

“yo!”

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);
}
```

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);  
}
```

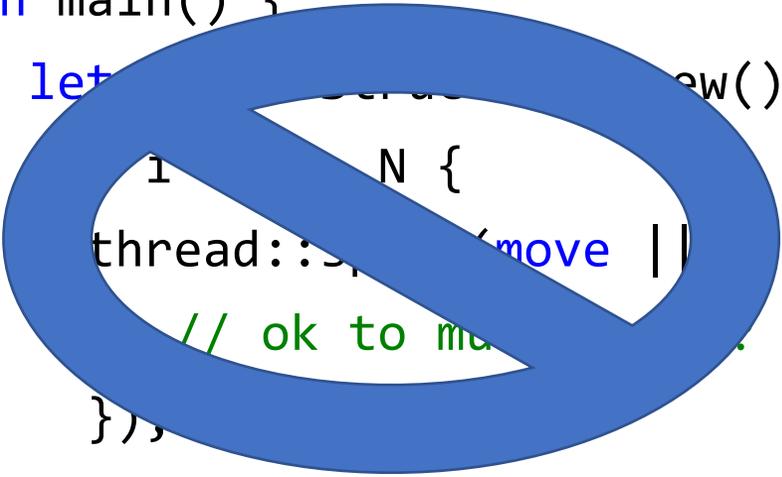
APIs return Option<T>

Sharing State

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

Sharing State

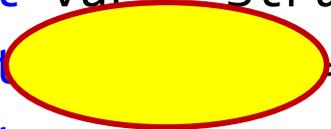
```
fn main() {  
    let server = Server::new();  
    for i in 1..N {  
        thread::spawn(move || {  
            // ok to mu  
        })  
    }  
}
```



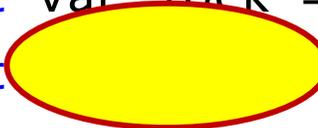
Sharing State: Arc and Mutex

```
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  = Mutex::new(var);  
    let var_arc = Arc::new(var);  
    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 = ();  
            // 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!

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!