Lifetimes, Function Types & More Ownership

Cooper Pierce & Jack Duvall

Carnegie Mellon University

Outline

Recall: Ownership Rules

Recall: Ownership Rules

- Every value has an "owner".

Recall: Ownership Rules

- Every value has an “owner”.
- There can only be one owner.

Recall: Ownership Rules

- Every value has an “owner”.
- There can only be one owner.
- When ownership of the value ends, the value will be “dropped”.

Recall: Ownership Rules

- Every value has an "owner".
- There can only be one owner.
- When ownership of the value ends, the value will be "dropped".
- You can have as many shared borrows (`&`) as you want, all at the same time ...

Recall: Ownership Rules

- Every value has an "owner".
- There can only be one owner.
- When ownership of the value ends, the value will be "dropped".
- You can have as many shared borrows (`&`) as you want, all at the same time ...
- ... but, you can only have one exclusive borrow (`&mut`), and not at the same time as any shared borrow.

Recall: Ownership Rules

- Every value has an “owner”.
- There can only be one owner.
- **When ownership of the value ends, the value will be “dropped”**.
- You can have as many shared borrows (`&`) as you want, all at the same time ...
- ... but, you can only have one exclusive borrow (`&mut`), and not at the same time as any shared borrow.

What's A Lifetime?

What's A Lifetime?

- Lifetime: "For a reference, the span of time that it can be used to accessed the underling value"

What's A Lifetime?

- Lifetime: "For a reference, the span of time that it can be used to accessed the underling value"
- Some subsection of the duration we can use the owning variable

What’s A Lifetime?

- Lifetime: “For a reference, the span of time that it can be used to accessed the underling value”
- Some subsection of the duration we can use the owning variable
- Construct of Rust’s borrow checker, not checked at runtime!

Lifetimes Roughly Correspond To Scope

```rust
// Error: x isn't in scope
let x_ref1 = &x;

let x = String::from("hello");

let x_ref2 = &x;
take_ownership(x);

// Error: x was moved
let x_ref3 = &x;
```

Recall: Returning An Invalid Reference

```rust
fn make_string() -> &String {
    let s = String::from("hello");
    &s
}
```

Recall: Returning An Invalid Reference

```rust
fn make_string() -> &String {
    let s = String::from("hello");
    &s
}
```

- Scope of `s` is the function body of `make_string`, which is the same as its lifetime

Recall: Returning An Invalid Reference

```
fn make_string() -> &String {
    let s = String::from("hello");
    &s
}
```

- Scope of `s` is the function body of `make_string`, which is the same as its lifetime
- Compiler knows lifetime of `make_string` will end once it returns, so reference won't be valid

Recall: Returning An Invalid Reference

```
fn make_string() -> &String {
    let s = String::from("hello");
    &s
}
```

- Scope of `s` is the function body of `make_string`, which is the same as its lifetime
- Compiler knows lifetime of `make_string` will end once it returns, so reference won't be valid
- (but first we'd run into an issue about what lifetime the returned reference would have)

Fixing The Example

Fixing The Example

Just don't return a reference! Move semantics already avoid copying things on the heap when not necessary[1]

```
fn make_string() -> String {
    String::from("hello")
}
```

Fixing The Example

Just don't return a reference! Move semantics already avoid copying things on the heap when not necessary[1]

```
fn make_string() -> String {
    String::from("hello")
}
```

[1] and the compiler will *automatically* determine if it's faster to pass pointer to output struct or pass via registers

Denoting Lifetimes

```
&'a Ty
&'a mut Ty
```

Denoting Lifetimes

```
&'a Ty
&'a mut Ty
```

- The `'a` is the lifetime name. The `'` is required, and the identifier can be any contiguous word.

Denoting Lifetimes

```
&'a Ty
&'a mut Ty
```

- The `'a` is the lifetime name. The `'` is required, and the identifier can be any contiguous word.
- The `'static` lifetime is special: denotes "will be valid until the program terminates"

Denoting Lifetimes

```
&'a Ty
&'a mut Ty
```

- The `'a` is the lifetime name. The `'` is required, and the identifier can be any contiguous word.
- The `'static` lifetime is special: denotes "will be valid until the program terminates"
- Not super common to need to denote explicitly, but sometimes necessary for:

Denoting Lifetimes

```
&'a Ty
&'a mut Ty
```

- The `'a` is the lifetime name. The `'` is required, and the identifier can be any contiguous word.
- The `'static` lifetime is special: denotes “will be valid until the program terminates”
- Not super common to need to denote explicitly, but sometimes necessary for:
  - Structs/Enums with references inside them

Denoting Lifetimes

```
&'a Ty
&'a mut Ty
```

- The `'a` is the lifetime name. The `'` is required, and the identifier can be any contiguous word.
- The `'static` lifetime is special: denotes "will be valid until the program terminates"
- Not super common to need to denote explicitly, but sometimes necessary for:
  - Structs/Enums with references inside them
  - Functions taking in those structs/enums

Denoting Lifetimes

```
&'a Ty
&'a mut Ty
```

- The `'a` is the lifetime name. The `'` is required, and the identifier can be any contiguous word.
- The `'static` lifetime is special: denotes "will be valid until the program terminates"
- Not super common to need to denote explicitly, but sometimes necessary for:
  - Structs/Enums with references inside them
  - Functions taking in those structs/enums
  - Other, more funky functions

Explicit Lifetimes In Structs

```
struct Vertex<'a> {
    edges: Vec<&'a Edge<'a>>,
}
struct Edge<'a> {
    info: EdgeInfo,
    vertex: &'a Vertex<'a>,
}
```

Explicit Lifetimes In Function Signatures

```
fn bfs<'a>(
    start_vertex: &'a Vertex<'a>,
    max_depth: usize,
) -> Vec<&'a Vertex<'a>> {
    ...
}
```

Returning An Invalid Reference Revisited

```
fn make_string<'a>() -> &'a String {
    let s = String::from("hello");
    &s
}
```

The same underlying issue as before, made more obvious by the lifetime annotation.

Rules For Lifetimes In Function Signatures

(From https://doc.rust-lang.org/rust-by-example/scope/lifetime/fn.html) Function signatures follow these rules:

Rules For Lifetimes In Function Signatures

(From https://doc.rust-lang.org/rust-by-example/scope/lifetime/fn.html) Function signatures follow these rules:

- Any reference *must* have an annotated lifetime

Rules For Lifetimes In Function Signatures

(From https://doc.rust-lang.org/rust-by-example/scope/lifetime/fn.html) Function signatures follow these rules:

- Any reference *must* have an annotated lifetime
- Any reference being returned *must* have the same lifetime as an input, or be `'static`

Rules For Lifetimes In Function Signatures

(From https://doc.rust-lang.org/rust-by-example/scope/lifetime/fn.html) Function signatures follow these rules:

- Any reference *must* have an annotated lifetime
- Any reference being returned *must* have the same lifetime as an input, or be `'static`

```
fn f1<'a, 'b>(x: &'a i32, y: &'b i32) -> &'a i32 {
    // what goes here?
}
```

Rules For Lifetimes In Function Signatures

(From https://doc.rust-lang.org/rust-by-example/scope/lifetime/fn.html) Function signatures follow these rules:

- Any reference *must* have an annotated lifetime
- Any reference being returned *must* have the same lifetime as an input, or be `'static`

```
fn f1<'a, 'b>(x: &'a i32, y: &'b i32) -> &'a i32 {
    // what goes here?
}
```

```
fn f2<'a, 'b>(x: &'a i32) -> &'b i32 {
    // what goes here?
}
```

Lifetime Elison

Certain patterns in Rust are very common:

```
// One input lifetime, return value is reference
fn f3<'a>(x: &'a i32) -> &'a i32 { ... }
// Multiple input lifetimes, return value is not reference
fn f4<'a, 'b, 'c>(x: &'a i32, y: &'b i32, z: &'c i32) -> i32 { ... }
```

Lifetime Elison

Certain patterns in Rust are very common:

```rust
// One input lifetime, return value is reference
fn f3<'a>(x: &'a i32) -> &'a i32 { ... }
// Multiple input lifetimes, return value is not reference
fn f4<'a, 'b, 'c>(x: &'a i32, y: &'b i32, z: &'c i32) -> i32 { ... }
```

So if a function signature falls into one of these patterns, you don't have to explicitly write lifetimes for it!

```rust
fn g3(x: &i32) -> &i32 { ... }
fn g4(x: &i32, y: &i32, z: &i32) -> i32 { ... }
```

Lifetime Elison Example

```
fn make_string(allocator: &mut Vec<String>) -> &String {
    allocator.push(String::from("hello"));
    &allocator[allocator.len() - 1]
}
```

Lifetime Elison Example

```
fn make_string(allocator: &mut Vec<String>) -> &String {
    allocator.push(String::from("hello"));
    &allocator[allocator.len() - 1]
}
```

- Input and output lifetimes elided to be the same

Lifetime Elison Example

```
fn make_string(allocator: &mut Vec<String>) -> &String {
    allocator.push(String::from("hello"));
    &allocator[allocator.len() - 1]
}
```

- Input and output lifetimes elided to be the same
- Valid reference returned via reference to original data

Sidenote: Loop Labels

```
'outer: for y in 0..5 {
    'inner: for x in 0..5 {
        if arr1[y][x] { break 'outer; }
        if arr2[x][y] { break 'inner; }
    }
}
```

Loop labels are not lifetimes—same syntax as lifetimes, and same sort of scope idea, but you can't actually make references with these names and have it make sense

Outline

What Is A Module?

What Is A Module?

- "A bag of things that go together"

What Is A Module?

- "A bag of things that go together"
  - Structs, Enums

What Is A Module?

- "A bag of things that go together"
  - Structs, Enums
  - Types, Traits

What Is A Module?

- "A bag of things that go together"
  - Structs, Enums
  - Types, Traits
  - Constants, Static members,

What Is A Module?

- "A bag of things that go together"
  - Structs, Enums
  - Types, Traits
  - Constants, Static members,
  - Other modules!

What Is A Module?

- “A bag of things that go together”
  - Structs, Enums
  - Types, Traits
  - Constants, Static members,
  - Other modules!
- Defines a namespace

Modules Within a File

```
fn f() { ... }
mod foo {
    fn f() { ... }
}
```

Directory Structure *Is* Module Structure

```
src/
├── lib.rs
└── bar/
    ├── mod.rs (bar)
    ├── baz.rs (bar::baz)
    └── qux.rs (bar::qux)
```

Directory Structure *Is* Module Structure

```
src/
├── lib.rs
└── bar/
    ├── mod.rs (bar)
    ├── baz.rs (bar::baz)
    └── qux.rs (bar::qux)
```

Alternatively,

```
src/
├── lib.rs
├── bar.rs (bar)
└── bar/
    ├── baz.rs (bar::baz)
    └── qux.rs (bar::qux)
```

Declaring File Modules

```rust
// In src/lib.rs:
mod bar;
```

Declaring File Modules

```rust
// In src/lib.rs:
mod bar;
```

```rust
// In the `bar` module:
mod baz;
mod qux;
```

Visibility

Visibility

By default, everything in a module is private to that module

Visibility

By default, everything in a module is private to that module

We need to explicitly declare items as public using the `pub` keyword:

```
pub struct Foo {
    x: usize,
    pub y: usize,
}
pub enum Bar {
    Bar1,
    Bar2,
}
pub fn calculate(f: Foo) -> Bar { ... }

pub mod baz;
mod qux;
```

Using Modules

```
mod foo {
    fn f() { ... }
}
fn main() {
    foo::f();
}
```

Using Modules

```
mod foo {
    fn f() { ... }
}
fn main() {
    foo::f();
}
```

Alternatively,

```
use foo::f;
fn main() {
    f();
}
```

Using Multiple Things At Once

```rust
use bar::{g, baz::h};
```

Using Multiple Things At Once

```rust
use bar::{g, baz::h};
```

```rust
use qux::*;
```

Using Multiple Things At Once

```rust
use bar::{g, baz::h};
```

```rust
use qux::*;
```

Useful for re-exports, collecting all useful includes into one “prelude”:

```rust
pub use crate::{
    bar::{g, baz::h},
    qux::*,
};
```

Outline

What Are Function Types?

What Are Function Types?

- Every value has a type

What Are Function Types?

- Every value has a type
- Functions are values! (sorry 15-122 stans)

What Are Function Types?

- Every value has a type
- Functions are values! (sorry 15-122 stans)
- Allows us to pass in functions as arguments to other functions, which many other good languages do in some capacity

Rust's Function Types

Rust's Function Types

- Function Pointers (sorry 15-150 stans): `fn (Ty1, Ty2, ...) -> Ty`

Rust's Function Types

- Function Pointers (sorry 15-150 stans): `fn (Ty1, Ty2, ...) -> Ty`
- Types Implementing Function Traits:

Rust's Function Types

- Function Pointers (sorry 15-150 stans): `fn (Ty1, Ty2, ...) -> Ty`
- Types Implementing Function Traits:
  - `Fn`

Rust's Function Types

- Function Pointers (sorry 15-150 stans): `fn (Ty1, Ty2, ...) -> Ty`
- Types Implementing Function Traits:
  - `Fn`
  - `FnOnce`

Rust's Function Types

- Function Pointers (sorry 15-150 stans): `fn (Ty1, Ty2, ...) -> Ty`
- Types Implementing Function Traits:
  - `Fn`
  - `FnOnce`
  - `FnMut`

What Is A Function Pointer?

Value of the function pointer type is either:

What Is A Function Pointer?

Value of the function pointer type is either:

- A “function item” (named function in the code), or

What Is A Function Pointer?

Value of the function pointer type is either:

- A "function item" (named function in the code), or
- A closure that doesn't capture (which is effectively the same)

Example: Using A Function Pointer

```
fn double(n: i32) -> i32 { 2 * n }
fn giveme_fnptr(f: fn(i32) -> i32) -> i32 {
    f(42)
}
fn test_fnptr() {
    assert_eq!(giveme_fnptr(double), 84);
}
```

Example: Using A Function Pointer

```
fn double(n: i32) -> i32 { 2 * n }
fn giveme_fnptr(f: fn(i32) -> i32) -> i32 {
    f(42)
}
fn test_fnptr() {
    assert_eq!(giveme_fnptr(double), 84);
}
```

Outline

Closure Syntax

From https://doc.rust-lang.org/book/ch13-01-closures.html

```
fn  add_one_v1   (x: i32) -> i32 { x + 1 }
let add_one_v2 = |x: i32| -> i32 { x + 1 };
let add_one_v3 = |x|             { x + 1 };
let add_one_v4 = |x|               x + 1  ;
```

Capturing State With Closures

If variable typed inside closure came from outside the closure, it is captured by reference

Capturing State With Closures

If variable typed inside closure came from outside the closure, it is captured by reference

- Immutable if possible, mutable if necessary

Capturing State With Closures

If variable typed inside closure came from outside the closure, it is captured by reference

- Immutable if possible, mutable if necessary

```rust
let z = 5;
let closure = |x| z == x;
```

Capturing State With Closures

If variable typed inside closure came from outside the closure, it is captured by reference

- Immutable if possible, mutable if necessary

```
let z = 5;
let closure = |x| z == x;
```

This can't be done with functions! Will fail to compile:

```
fn f(x: i32) -> bool { z == x }
```

Consuming State With Closures

Sometimes, we *do* want to move a value into a closure:

```
let message = String::from("hello");
thread::spawn(move || {
    println!("{}", message);
});
```

Consuming State With Closures

Sometimes, we *do* want to move a value into a closure:

```
let message = String::from("hello");
thread::spawn(move || {
    println!("{}", message);
});
```

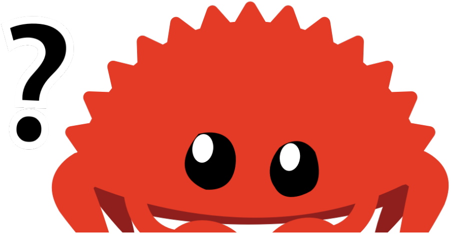
`move` keyword: anything that would be captured by reference is now captured by value (moved)

Things Closures Can't Be

- Recursive
- Generic
- In most cases, function pointers
  - If a closure doesn't capture anything from its environment, it can be coerced to a function pointer:

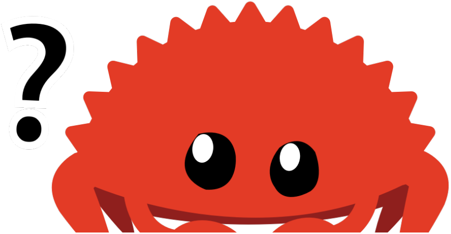
```
let x: fn(i32, i32) -> i32 = |x, y| x + y;
```

Type Of A Closure



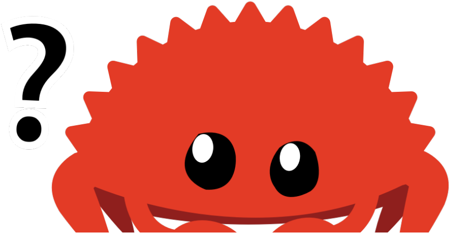
Type Of A Closure

- You can't write down their type!



Type Of A Closure

- You can't write down their type!
- Wait, so how can we take them as arguments??



Outline

Traits Aren't Types

Traits Aren't Types

- Types: correspond to the compiler's representation of data

Traits Aren't Types

- Types: correspond to the compiler's representation of data
- Traits: describe what a type can do

Fn Trait

```
let fn_closure = |x| 2 * x;
```

`Fn` Trait

```
let fn_closure = |x| 2 * x;
```

- We say: `fn_closure` implements `Fn(i32) -> i32`

`Fn` Trait

```
let fn_closure = |x| 2 * x;
```

- We say: `fn_closure` implements `Fn(i32) -> i32`
- Can be called by shared reference

`Fn` Trait

```
let fn_closure = |x| 2 * x;
```

- We say: `fn_closure` implements `Fn(i32) -> i32`
- Can be called by shared reference
- Closure must:

Fn Trait

```
let fn_closure = |x| 2 * x;
```

- We say: `fn_closure` implements `Fn(i32) -> i32`
- Can be called by shared reference
- Closure must:
  - Not mutate any captured state

`Fn` Trait

```
let fn_closure = |x| 2 * x;
```

- We say: `fn_closure` implements `Fn(i32) -> i32`
- Can be called by shared reference
- Closure must:
  - Not mutate any captured state
  - Not move any captured state out

`Fn` Trait

```
let fn_closure = |x| 2 * x;
```

- We say: `fn_closure` implements `Fn(i32) -> i32`
- Can be called by shared reference
- Closure must:
  - Not mutate any captured state
  - Not move any captured state out
- All (safe) function pointers also implement `Fn`

Example: Using `Fn`

```
fn giveme_fn1(f: impl Fn(i32) -> i32) -> i32 {
    f(42)
}
// Or, verbosely:
fn giveme_fn2<T: Fn(i32) -> i32>(f: T) -> i32 {
    f(42)
}
// Or, even more verbosely:
fn giveme_fn3<T>(f: T) -> i32
    where T: Fn(i32) -> i32
{
    f(42)
}
```

Example: Using Fn

```rust
fn giveme_fn1(f: impl Fn(i32) -> i32) -> i32 {
    f(42)
}
// Or, verbosely:
fn giveme_fn2<T: Fn(i32) -> i32>(f: T) -> i32 {
    f(42)
}
// Or, even more verbosely:
fn giveme_fn3<T>(f: T) -> i32
    where T: Fn(i32) -> i32
{
    f(42)
}
```

Example: Using Fn

```
fn giveme_fn1(f: impl Fn(i32) -> i32) -> i32 {
    f(42)
}
// Or, verbosely:
fn giveme_fn2<T: Fn(i32) -> i32>(f: T) -> i32 {
    f(42)
}
// Or, even more verbosely:
fn giveme_fn3<T>(f: T) -> i32
    where T: Fn(i32) -> i32
{
    f(42)
}
```

Example: Using Fn

```
fn giveme_fn1(f: impl Fn(i32) -> i32) -> i32 {
    f(42)
}
// Or, verbosely:
fn giveme_fn2<T: Fn(i32) -> i32>(f: T) -> i32 {
    f(42)
}
// Or, even more verbosely:
fn giveme_fn3<T>(f: T) -> i32
    where T: Fn(i32) -> i32
{
    f(42)
}
```

FnMut Trait

```
let mut state = 0;
let fnmut_closure = |x| {
    state += x;
    state
};
```

FnMut Trait

```
let mut state = 0;
let fnmut_closure = |x| {
    state += x;
    state
};
```

- Can be called by mutable reference

FnMut Trait

```
let mut state = 0;
let fnmut_closure = |x| {
    state += x;
    state
};
```

- Can be called by mutable reference
- Closure must not move any captured state out

Example: Using FnMut

```
fn giveme_fnmut(mut f: impl FnMut(i32) -> i32) -> i32 {
    let x = f(42);
    f(x)
}
assert_eq!(giveme_fnmut(fnmut_closure), 84);
```

Example: Using FnMut

```
fn giveme_fnmut(mut f: impl FnMut(i32) -> i32) -> i32 {
    let x = f(42);
    f(x)
}
assert_eq!(giveme_fnmut(fnmut_closure), 84);
```

FnOnce Trait

```
let state = Box::new(42);
let fnonce_closure = move |x| {
    let y = x + *state;
    drop(state);
    y
};
```

FnOnce Trait

```
let state = Box::new(42);
let fnonce_closure = move |x| {
    let y = x + *state;
    drop(state);
    y
};
```

- Can be called by taking ownership of the closure

FnOnce Trait

```
let state = Box::new(42);
let fnonce_closure = move |x| {
    let y = x + *state;
    drop(state);
    y
};
```

- Can be called by taking ownership of the closure
- All closures implement this

Example: Using `FnOnce`

```
fn giveme_fnonce(f: impl FnOnce(i32) -> i32) -> i32 {
    let x = f(42);
    // let y = f(9 * 6); // Does not compile
    x
}
```

Why Are There So Many Different Traits??

Why Are There So Many Different Traits??

- Need to distinguish between all the different ways we can capture state, interact with borrow/ownership system!

Why Are There So Many Different Traits??

- Need to distinguish between all the different ways we can capture state, interact with borrow/ownership system!
  - `Fn`: "This acts like a function pointer, doesn't modify any local state"

Why Are There So Many Different Traits??

- Need to distinguish between all the different ways we can capture state, interact with borrow/ownership system!
  - `Fn`: "This acts like a function pointer, doesn't modify any local state"
  - `FnMut`: "This may modify local state, but doesn't result in any local state being dropped when called"

Why Are There So Many Different Traits??

- Need to distinguish between all the different ways we can capture state, interact with borrow/ownership system!
  - `Fn`: "This acts like a function pointer, doesn't modify any local state"
  - `FnMut`: "This may modify local state, but doesn't result in any local state being dropped when called"
  - `FnOnce`: "This can only be called 0 or 1 times because it may drop local state when called."

Why Are There So Many Different Traits??

- Need to distinguish between all the different ways we can capture state, interact with borrow/ownership system!
  - `Fn`: "This acts like a function pointer, doesn't modify any local state"
  - `FnMut`: "This may modify local state, but doesn't result in any local state being dropped when called"
  - `FnOnce`: "This can only be called 0 or 1 times because it may drop local state when called."
- Anything higher on the list can be used as anything lower on the list

Manually Implementing Function Traits?

Unfortunately, only on nightly, a.k.a. “unstable” Rust. Only closures will implement these traits for now.

Next Time

- Livecoding!!

Homework: What We Meant To Give You Last Time

Tarball: https://rust-stuco.github.io/handouts/TODO-handout.tgz
Handout PDF: https://rust-stuco.github.io/handouts/TODO-writeup.pdf