Ownership and Function Types

also to include lifetimes, more borrow checker rules, and closures

Jack Duvall

Carnegie Mellon University

Ownership

Recall: Stack and Heap

- Regions of memory you can store data in
- Stack:
  - Local to current function invocation
  - Data must have known size at compile time
  - Automatically freed when function exits
- Heap:
  - Entire program can view
  - Data can have unknown size
  - Must allocate and free ”manually”

Definitions

- Value: "The literal bits in memory somewhere"
- Variable: "The label for those bits at any given moment"

```
// The variable x has a value of 98008
let x = 98008;
```

More Definitions

- Scope: "A set of `{}`"
- Dropping: "Making a value inaccessible"
  - e.g. popping stack frame or calling `free()`

```
fn f() {  // x is scoped to f
  let x = String::from("hello");
  drop(x); // x is manually dropped
}
```

Ownership Rules

- From https://doc.rust-lang.org/stable/book/ch04-01-what-is-ownership.html
- Each value in Rust has a single variable called its owner.
- There can only be one owner at a time.
- When the owner exits its scope, the value will be dropped.

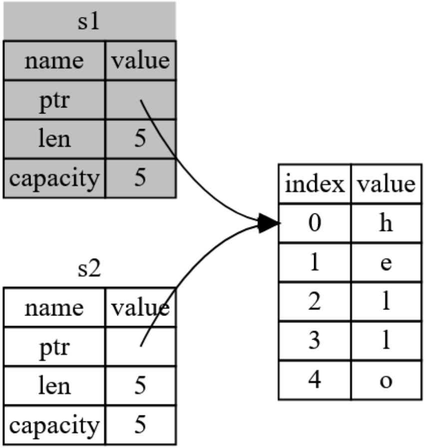
Upcoming Ownership Examples

- Simple Move
- Simple Copy
- Move Into Function
- Copy Into Function
- Cloning

Ownership Example: Simple Move

```rust
let s1 = String::from("hello");
let s2 = s1; // `s2` now "owns" the data that `s1` used to refer to
println!("{}", s1); // So this is an error
```

Ownership Example: Simple Move



Ownership Example: Simple Copy

```
let x = 5;
let y = x; // `x` can be copied efficiently, so the data is just
           // copied into `y`
println!("{}", x); // This is OK
```

Ownership Example: Move Into Function

```rust
fn take_ownership(y: String) { println!("{}", y); }
fn main() {
    let x = String::from("hello");
    take_ownership(x);
    // using `x` is an error here, because `take_ownership` took
    // ownership, so `x`'s value is somewhere else
}
```

Ownership Example: Copy Into Function

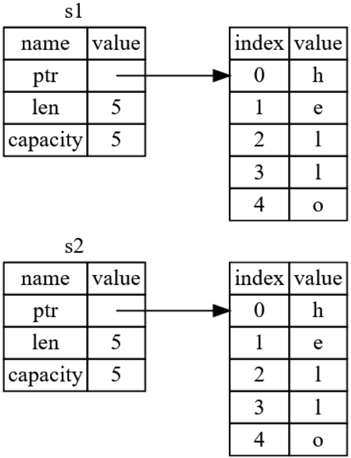
```rust
fn makes_copy(y: i32) { println!("{}", y); }
fn main() {
    let x = 5;
    makes_copy(x);
    // Passing `x` into `makes_copy` made a copy of `x`'s value,
    // so `x` still has ownership
}
```

Ownership: Cloning

- What if you have data that can't be automatically copied, but you still want a copy?
- Solution: `.clone()` the data!

```rust
let s1 = String::from("hello");
let s2 = s1.clone();
// `s1` and `s2` refer to different memory locations now
```

Ownership: Cloning: Diagram



When Can I Copy Or Clone?

- Copy: whenever a type implements the `Copy` trait!
- Clone: whenever a type implements the `Clone` trait!
- We'll get into traits more next lecture
- Important: the programmer implementing the struct decides if/how these operations are allowed
  - Restriction on `Copy`: every field/variant must be `Copy`
  - If something is `Copy`, it must also be `Clone`

Borrowing

References: Pointers But Better

- Reference: “You don’t own this value, but you can still access it”
  - Value is called ”borrowed”
- Two types: Immutable and Mutable
- Guarantee: it’s always valid to access memory through a reference!
  - Not the case with pointers

Immutable References

```
&Ty
```

- Only let you read
- Any number can exist at one point, so long as there's no mutable references to the object at the same time.

Immutable References: Example

```
let x: i32 = 5;
let x_ref: &i32 = &x;
let x_ref2: &i32 = &x; // Ok to have more than one immutable ref
let x_ref3: &i32 = x_ref; // Immutable reference is `Copy`
let y: i32 = *x_ref; // Ok, `i32` is `Copy`
```

Mutable References

```
&mut Ty
```

- Let you read and write
- Can only be made if the underlying object is also mutable
- Only one can exist at a time

Mutable References: Example

```
let x: i32 = 5;
let x_mut_ref: &mut i32 = &mut x; // Does not compile, `x` is not
                                   // `mut`

let mut y: i32 = 6;
let y_mut_ref: &mut i32 = &mut y;
let y_mut_ref2: &mut i32 = &mut y; // Does not compile, can't
                                    // have more than one mut ref
let y_mut_ref3: &mut i32 = y_mut_ref; // Does not compile, mut
                                       // refs aren't `Copy`

*y_mut_ref += 2;
```

Lifetimes

Why Do We Need Lifetimes?

- To know how long a reference is valid for!
- Lifetime: "For a variable, the span of time that it owns a value"
- Roughly corresponds to the scope of the variable
- Construct of Rust's borrow checker, not checked at runtime!

Lifetimes Roughly Correspond To Scope

```rust
fn main() {
    let x_ref1 = &x; // Does not compile, can't reference `x`
                     // before it's defined
    let x = String::from("hello");
    let x_ref2 = &x;
    take_ownership(x);
    let x_ref3 = &x; // Does not compile, `x`'s value has been
                     // moved out, no longer in this scope
}
```

Returning 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

Fixing The Example: Use Moves

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

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

Denoting Lifetimes

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

- The `'a` is the lifetime name. The `'` is required, and the identifier can be any contiguous word[1].
- The `'static` lifetime is special: denotes "will be valid until the program terminates"
- Rare you'll need to denote explicitly, but sometimes necessary for:
  - Structs/Enums with references inside them
  - Functions taking in those structs/enums
  - Other, more funky functions

[1]Looking at you, SML

Explicit Lifetimes In Structs

```rust
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>> {
    ...
}
```

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

Wait, didn't we forget to write these explicit lifetimes last lecture??

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 { ... }
```

So if it falls into one of these patterns, you don't have to explicitly write them

Fixing The Example Again: Allocators

```
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

Not Actually Lifetimes: 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; }
    }
}
```

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

Function Types

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

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

Function Pointers

Attributes Of 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);
}
```

Closures

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

- 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);
});
```

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

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

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

Function Traits

Traits Aren't Types

- Types: correspond to the compiler's representation of data
- Traits: describe what a type can do
- More about this next lecture

`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

```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`

```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

```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`

```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)
}
```

`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
};
```

- 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??

- 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

Homework

Function Type Puzzle

https://github.com/Rust-Stuco/puzzles/tree/main/03_function_types