Lecture 2: Ownership

Ownership, Result & Option, and Error Handling

Pollev time

pollev.com/joyliu761

What is something that you are confused about from last lecture or the ownership book chapter?

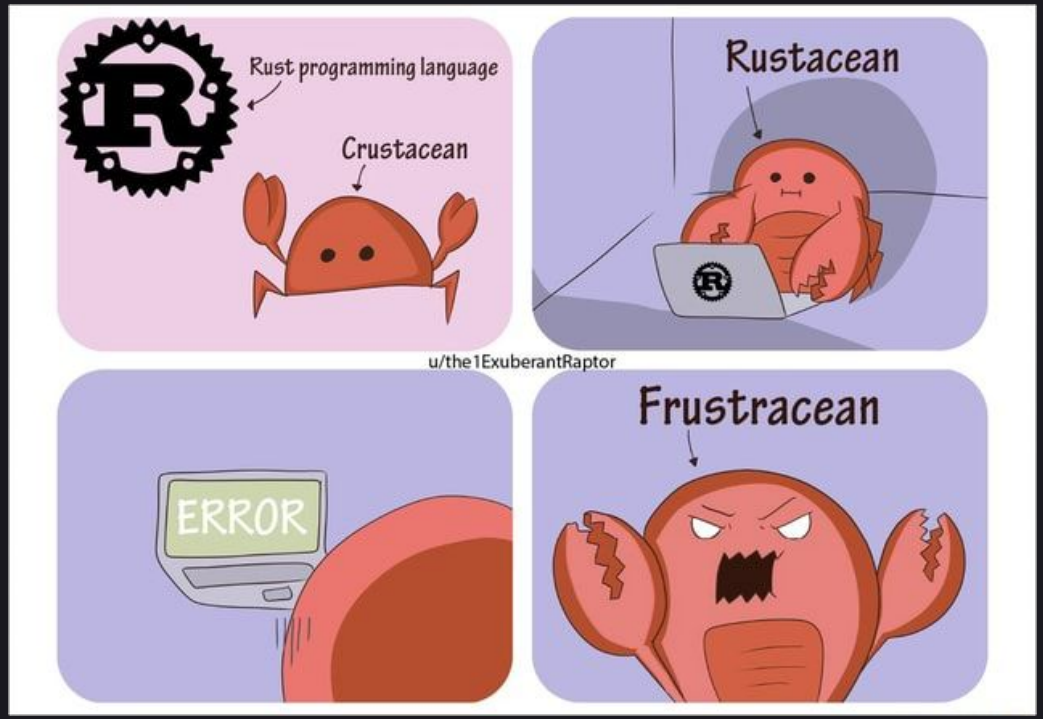
Logistics

HW1

- Due Sunday Feb 4th
- Available on Ed

Office Hours

- Sunday 5-7pm
- Thursday 12:30-2:30pm
- Rooms posted on Ed



Aside: Git Crash Course?

- As per iOS / NETS2120 head TA request :)
  - Maybe recorded?
- During OH if we have folks who are shaky on git

```
dan@Daniels-MacBook-Pro ~/Sites/merge_conflicts_example (example-branch) $ git merge develop
Auto-merging example.php
CONFLICT (content): Merge conflict in example.php
Automatic merge failed; fix conflicts and then commit the result.
dan@Daniels-MacBook-Pro ~/Sites/merge_conflicts_example (example-branch *+|MERGING) $ git status
On branch example-branch
You have unmerged paths.
  (fix conflicts and run "git commit")

Unmerged paths:
  (use "git add <file>..." to mark resolution)

        both modified:   example.php

no changes added to commit (use "git add" and/or "git commit -a")
dan@Daniels-MacBook-Pro ~/Sites/merge_conflicts_example (example-branch *+|MERGING) $
```

Takeaways

Last week

- Strong types: every variable has a type
  - Rust built-in data types and primitives
  - Structs and Enums

This week

- Intermediate Rust data types
  - Strings and collections
  - Result/Option & Error Handling
- **Rust uses a set of ownership rules checked in compile time to manage memory**

Result/Options

Options

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

Result

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

Type of error may vary:

- **`Result<usize, std::io::Error>`**
- **`Result<f32, std::err::Error>`** (good practice)
- `Result<MyStruct, &'static str>` (lazy)

Result → Option using `.ok()`

```
let res: Result<u8, ()> = Ok(42);
let opt: Option<u8> = res.ok();
```

Deconstructing Result/Options
(and enums in general)

if/let syntax + match statements: safe and convenient destructuring

```
if let Some(x) = my_option {
    println!("the inner value is {}", x);
} else {
    println!("the value does not exist");
}
match my_option {
   Some(x) -> // do smth
   None -> // do smth
}
```

Error Handling

Error Handling in Rust

Rust encourages (mandates) explicit error handling

1. **`match`** and **`if let`** for handling **`Result<T, E>`**
2. Using the **`?`** operator to propagate errors
3. **`unwrap`**/**`unwrap_or`**/**`unwrap_or_else`** to force extract value, panic (raise exception) if is error

```
match my_result {
     Ok(x) => println!("x is {}", x),
     Err(e) => eprintln!("error occurred: ({})", e),
};
```

Error Handling Patterns

Result

- Early returns with 'Result' and ‘?’ operator
- Error chaining for complex operations
- Recoverable vs. Unrecoverable errors
- *Make it obvious to the caller that a function can fail*

The 'panic!' Macro

- Handling unrecoverable errors with panics
- Unwinding and aborting the program
- Use when you are in an unrecoverable situation

```
fn func_1(x: Input) -> Result<Data1, MyErr> { ... }
fn func_2(x: Data1) -> Result<Data2, MyErr> { ... }
fn func_3(x: Data2) -> Result<Output, MyErr> { ... }

fn f(input: Input) -> Result<Output, MyErr> {
      let data_1 = func_1(input)?;
      let data_2 = func_2(data_1)?;
      func_3(data_2)
}
```

What happens here?

```
fn func_1(x: Input) -> Result<Data1, std::io::Error> { ... }
fn func_2(x: Data1) -> Result<Data2, MyErr> { ... }
fn func_3(x: Data2) -> Result<Output, &'static str> { ... }

fn f(input: Input) -> Result<Output, (?????)> {
    let data_1 = func_1(input)?;
    let data_2 = func_2(data_1)?;
    func_3(data_2)
}
```

Extern Crate: anyhow

Type of error may vary:

- `Result<usize, std::io::Error>`
- `Result<f32, std::err::Error>` (good practice)
- `Result<MyStruct, &'static str>` (lazy)

```
fn string_error() -> Result<(), String> {
    Ok(())
}
fn io_error() -> Result<(), std::io::Error> {
    Ok(())
}
fn any_error() -> Result<(), Box<dyn Error>> {
    string_error()?;
    io_error()?;
    Ok(())
}
```

Err(anyhow!(“error message”))

anyhow::Result is alias for Result<T, anyhow::Error>

Context / with_context function - provide more information on top of error

```rust
fn string_error() -> Result<(), String> {
    Ok(())
}
fn io_error() -> Result<(),
std::io::Error> {
    Ok(())
}
fn any_error() -> Result<(), Box<dyn
Error>> {
    string_error()?;
    io_error()?;
    Ok(())
}
```

```rust
use anyhow::Result;

fn string_error() -> Result<()> {
    Ok(())
}

fn io_error() -> Result<()> {
    Ok(())
}

fn any_error() -> Result<()> {
    string_error()?;
    io_error()?;
    Ok(())
}
```

Strings and Collections

Difference between Vec and [T; N]

- For any type T, you can make an array `[T; N]`
  - For example, `[i32; 4]` is a 4-length i32 array
  - The length of the array is fixed at compile time
    i. That is, it cannot be changed, and cannot depend on a variable
  - Lives on the stack
- For *growable* arrays, or arrays which we don't know the size of at compile time, we need to use `Vec<T>`
  - Lives on the heap
- Slice
  - &vector[0..5]
  - &array[1...3]

Growable arrays (Vec struct)

- `pub fn len(&self) -> usize`
- `pub fn pop(&mut self) -> Option<T>`
- `pub fn push(&mut self, value: T)`
- `fn into_iter(self) -> <Vec<T, A> as IntoIterator>::IntoIter`
  - Creates a consuming iterator, that is, one that moves each value out of the vector (from start to end). The vector cannot be used after calling this.

Difference between &str and String

- You can make an &str as follows - `“hi”`
  - The length of &str is fixed at compile time
    i. That is, it cannot be changed, and cannot depend on a variable
  - &str “points” to data
    i. If it “points” to an original String, data it points to lives on the heap
    ii. If it “points” to a string literal (“hello”), data is stored in the binary
    * iii. str → [char]
    iv. &str → &[char]
- For mutable String with unknown size at compile time, we need to use **`String`** object
  - Lives on the heap

*The data is UTF-8 encoded bytes.

Ownership

Memory Management is hard.

Memory Management is hard

- Accessing values stored on the heap
- Mutable values
- Remember to free memory at the end
- Etc.

Issues

- Duplication
- Data races or inconsistencies
- Memory leaks
- Undefined behavior
- Dangling pointers

Aside: **Garbage collectors**

- Scans the heap with a **mark-and-sweep** algorithm to find objects no longer in use, then remove the object, freeing up memory.
- This process continues until all unused objects are successfully reclaimed.

Sometimes, a developer will inadvertently write code that continues to be referenced even though it’s no longer being used. The garbage collector will not remove objects that are being referenced in this way, leading to memory leaks.

Memory Mgmt Best Practices

- Each value is only stored in one place.
- Can’t change the value from multiple points.
- When you don’t need it, set it free

Garbage collectors

Recommend Programming Patterns (Wild West)

Compile Time Static Analysis

Memory Mgmt Best Practices

Not just best practices, parts of them are encoded in the C standard!

“An object has a storage duration that determines its lifetime. There are three storage durations: static, automatic, and allocated… If an object is referred to outside of its lifetime, the behavior is undefined.”
C99 standard

Rust Ownership Rules

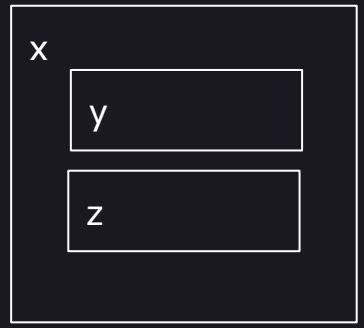
Ownership of a value is a static, syntactic property

1. Each value in Rust has a single owner
2. There can only be one owner at a time
3. When the owner goes out of scope, the value will be dropped

**This is actually gospel truth*

Scope

Range within a program for which an item is valid




```
fn main() {
    let x = 10; // Variable x is in the outermost scope
    {
        let y = 20; // Variable y is in an inner scope
        println!("Inside inner scope: x = {}, y = {}", x, y);
    } // y goes out of scope here
    {
        let z = 30; // Variable z is in another inner scope
        println!("Inside another inner scope: x = {}, z = {}", x, z);
    } // z goes out of scope here
    // You can still access x here, but y and z are no longer accessible
    println!("Outside all scopes: x = {}", x);
}
```

What are the ways you interact with data?

Levels of access

Ownership Rules

1. Each value in Rust has a single owner
2. There can only be one owner at a time
3. When the owner goes out of scope, the value will be dropped

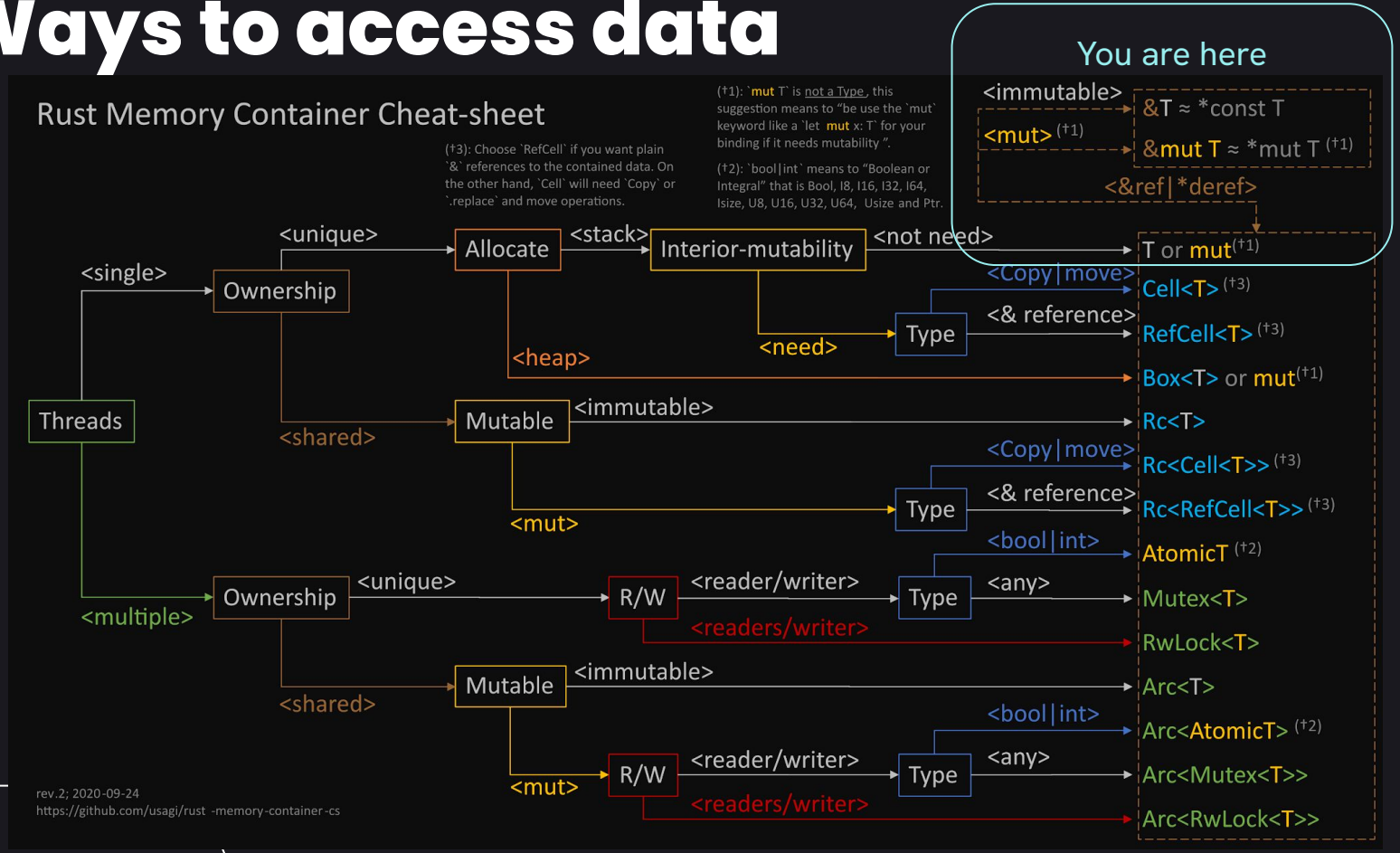
Ways to access data

- Own (Move)
  - Consume the data: I don’t need the original value anymore
- Copy/Clone
  - Duplicate the data
- Borrow
  - Just read the data
  - Read and change the data

Ownership Rules

1. Each value in Rust has a single owner
2. There can only be one owner at a time
3. When the owner goes out of scope, the value will be dropped

Ways to access data



Move: Transferring Ownership

Rust avoids dangling pointers by ensuring one owner at a time, so how do you transfer ownership?

- Move of Owned Values (easily Copy-able types)

```
let x = 5;
let y = x;
```

- Move of Non-Copy Types

```
let s1 = String::from("hello");
let s2 = s1;
```

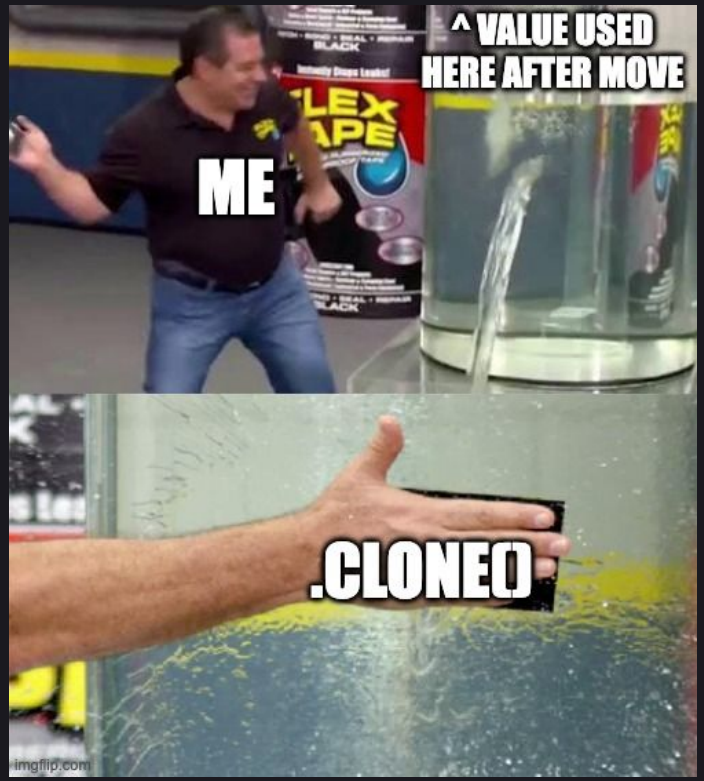
- Move of Ownership in Function Calls
- Returning Ownership from Functions

Clone

Before move, you just clone the value:

```
let s1 = String::from("hello");
let s2 = s1.clone()
```

This deep copies the heap data



Borrow

- **References (&) and Dereferences (*)**
- Borrowing rules:
  - There can be more than one reference to a resource
  - There can be only one mutable reference to a resource at the time
  - Any borrow may not last longer than a scope of the owner

Ownership Rules

1. Each value in Rust has a single owner
2. There can only be one owner at a time
3. When the owner goes out of scope, the value will be dropped

What is a Reference?

What if you were programming in C, but you had the promise that every pointer you use was

1. Never null. Null pointer dereference
2. Never dangling. Use after free
3. Always aligned. Slow reads (x86) and crashes (ARM)
4. Always pointing to a valid value. Invalid values (bool of value 2, enums out of range)

How does ownership provide Reference guarantees?

1. Never null. A value's owner always stored the data, so it was never null.

https://play.rust-lang.org/?version=stable&mode=debug&edition=2021&gist=7e507b1edfb3f040d70e3bb26699546c

How does ownership provide Reference guarantees?

2. Never dangling. A value's owner always stores its data, and you can never use the reference after the owner has gone out of scope.

https://play.rust-lang.org/?version=stable&mode=debug&edition=2021&gist=fd1075f06888bb6c9f533f8b1b6052f3

How does ownership provide Reference guarantees?

3. Always aligned. The owner had to store the data in a valid place, so a pointer to it must be aligned.

Some CPUs punish unaligned reads by making them slower. Other CPUs disallow unaligned reads altogether, and they will cause faults/crashes.

Compiled languages (including C!) assume their reads are aligned and try to avoid running into this.

(optional asm demo) https://godbolt.org/z/4cjzb7zz5

How does ownership provide Reference guarantees?

4. Always points to a valid value. The owner had to store a valid value of the type, so the reference to that owner must point to a valid value.

https://play.rust-lang.org/?version=stable&mode=debug&edition=2021&gist=b557c7e6fb46b0e84d7f1593bbf4dbe8

Borrowing Rust Playground

- Rust Playground

Rust Ownership, in Summary

- Borrow
  - Just read the data - **Reference (&T)**
  - Read and change the data - **Mutable Reference (&mut T)**
- Copy/Clone the data
  - Doesn't affect original value (**.clone()**)
- Own the data
  - I don't need the original value anymore:
    i. **Move it**

Rust Ownership Playground

Your turn: pollev.com/joyliu761

Option Ownerships

`take()`: In Rust, the take() method takes ownership of an Option's contained value, replacing it with None. For example:

```
let my_option = Some(10);
if let inner_value = my_option.take() {
  // do something here
}
assert_eq!(my_option, None);
```

Option Ownerships

```
let mut value = Some(“hi”);

if let Some(ref mut inner_value) = value {
  *inner_value = “sup”;
}
```

By default, if let takes ownership of the value. By using the `ref mut` keyword, you are instead creating a mutable reference so you can modify it without consuming the value.

What error message if you delete the ref mut? What if you delete just ref? What if you replace `ref mut` with `&mut`?

Option Ownerships (match ergonomics)

```
let mut value = Some("hi");

if let Some(inner_value) = &mut value {
  *inner_value = "sup";
}
```

If we pattern match on a reference to a value, we get the same kind of reference to the insides of that value. This is called "match ergonomics," and it's the same as the compiler inserting the `ref mut` keyword we used before.

This help from the compiler is all or nothing. If you want to use match ergonomics for one part of a pattern, make sure that the rest of the pattern doesn't have manual uses of `ref` or `mut` around.

Your Turn

tinyurl.com/cis1905-lec2-practice