Lecture 5

Immutability

Substructural Type System

- The reason we make type systems is to help us solve actual problems.

1. can be used any number of times
2. can't be used more than once (affine)
3. must be used at least once
4. must be used exactly once (linear)

1- Can Be Used Any Number Of Times

- Rust provides two forms:

Copy

- total unrestricted use of the value.
- Copying is done by a bitwise copy.
- Discarding is done by forgetting about it.
- Plain Old Data types like i32 and f64 are Copy.

Clone

- like Copy but needs some work when we do the copy or destroy.
- Copying a Clone value requires an explicit call to clone().
- Destroying a Clone value implicitly calls drop().
- Almost every type in Rust is Clone.

2- Can't Be Used More Than Once

```
let x = Vec::new();
x.push(1);
x.push(2);
let y = x;             // NOTE: x moved here
println!("{:?}", x); // ERROR: use of moved value x
pritnln!("{:?}", y); // OK
```

- This is what Rust calls move semantics, or move-only types.
- Rust defines a "use" to be pass-by-value.
- Pass-by-reference isn't considered a use, Why?.
- Proves that we have unique access to a value.
- Gives strong aliasing guarantees.
- Avoids logic bugs.

3- Must Be Used At Least Once

- The form Rust has the loosest support for.
- Can be seen in two places:

1. the *unused_variables*, *unused_assignments*, and *unused_must_use* lints
2. Drop (Deferred until Slide 16)

unused_variables

```
fn main() {
    let x = 0; // WARNING: unused variable `x`
    let y = 1;
    if y == y { println!("OK!") }
}
```

- basically any use of the identifier x will silence the lint

3- Must Be Used At Least Once

unused_assignments

```
fn main() {
 let mut x = 0; // WARNING: value of `x` is never read
 x = 1;
 println!("{}", x);
}
```

- checks to see if there are any assignments that always get overwritten before a read.

3- Must Be Used At Least Once

unused_must_use lints

```
fn read() -> Result<i32, i32> { Ok(0) }

fn main() {
    read();    // WARNING: unused result which must be used
    let _ = read(); // OK
}
```

4- Must Be Used Exactly Once

- It's just a move-only must-use type.
- It is Case 2 and Case 3.

Immutability in RUST

Rust has chosen to make the immutable case the default.

Why should everything be marked immutable?

1- Readability

- You just know that once a variable has been given a value, it remains that way.

2- No mistakes with APIs

- Projects grow and it’s hard to track all the interactions between components.
- In those cases, it’s very useful to know that calling a specific function won’t mutate its arguments.

Immutability in RUST

3- Better designs? Even in Python

```
adults = []
for person in people:
    if person.age >= 21:
        adults.append(person)
```

```
adults = [person for person in people if person.age >= 21]
# OR
adults = filter(lambda person: person.age >= 21, people)
```

- Reasoning over a loop is hard: loops can do “anything”, so you must read the whole structure to understand what is going on and what the side-effects are. On the other hand, these two one-liners show exactly what your intent is.

4- Concurrency!!!!!!

- Immutable data structures are trivial to parallelize.

The Borrow Checker

- The borrow checker is the component in the Rust compiler that enforces data ownership rules.
- It enforces these rules to prevent data races.
- What is a **data race**?

There is a “data race” when two or more pointers access the same memory location at the same time, where at least one of them is writing, and the operations are not synchronized.

How to prevent Data Race?

The borrow checker rules:

1- Each value in Rust has a variable that is called its owner.

2- There can only be one owner at a time.

3- When the owner goes out of scope, the value is dropped.

How to prevent Data Race? and other Bugs!

(Ownership)

Simple Example

```
#[derive(Debug)]
struct Eggs(i32);

fn uses_eggs(eggs : Eggs ) {
    println!("I consumed {:?}", eggs);
}

fn main() {
    let x = Eggs(1);
    uses_eggs(x);
}
```

- #[derive(...)] required to make this `struct` printable with `fmt::Debug`.
- {:?} means "use the Debug trait for displaying this"
- Eggs is a new type wrapper.
- In the main function, x contains an Egg.
- When it calls uses_eggs, ownership of that Eggs passes to uses_eggs.

Simple Example

- Using x again in main would give an error.

```
fn main() {
    let x = Eggs(1);
    uses_eggs(x);
    uses_eggs(x);
}
```

- Output:

```
error[E0382]: use of moved value: `x`
  --> src\main.rs:11:15
   |
9  |     let x = Eggs(1);
   |         - move occurs because `x` has type `Eggs`,
which does not implement the `Copy` trait
10 |     uses_eggs(x);
   |               - value moved here
11 |     uses_eggs(x);
   |               ^ value used here after move
```

Dropping

When a value goes out of scope, the memory is freed.

```
#[derive(Debug)]
struct Eggs(i32);
impl Drop for Eggs {
    fn drop(&mut self) {
        println!("Dropping eggs: {:?}", self);
    }
}
fn uses_eggs(eggs: Eggs) {
    println!("I consumed: {:?}", eggs);
}                                       // Drop is run here
fn main() {
    let x = Eggs(1);
    println!("Before uses_eggs");
    uses_eggs(x);
    println!("After uses_eggs");
}
```

Question: What is the output of the program?

Dropping

Output:

```
Before uses_eggs
I consumed: Eggs(1)
Dropping eggs: Eggs(1)
After uses_eggs
```

Lexical scoping

```
#[derive(Debug)]
struct Eggs(i32);

impl Drop for Eggs {
    fn drop(&mut self) {
        println!("Dropping Eggs: {:?}", self);
    }
}
fn main() {
    println!("Before x");
    let _x = Eggs(1);
    println!("After x");
    {                                   //Block = function
        println!("Before y");
        let _y = Eggs(2);
        println!("After y");
    }                                   // Drop y is run here
    println!("End of main");
}                                       // Drop x is run here
```

Lexical scoping

The output from this program is:

```
Before x
After x
Before y
After y
Dropping Eggs: Eggs(2)
End of main
Dropping Eggs: Eggs(1)
```

Borrows/references (immutable)

What if we want to share a reference to a value without moving ownership?

```
#[derive(Debug)]
struct Eggs(i32);
impl Drop for Eggs {
    fn drop(&mut self) {
        println!("Dropping Eggs: {:?}", self);
    }
}
fn uses_eggs(eggs: &Eggs) {
    println!("I consumed : {:?}", eggs);
}
fn main() {
    let x = Eggs(1);
    println!("Before uses_eggs");
    uses_eggs(&x);
    uses_eggs(&x);
    println!("After uses_eggs");
}
```

Borrows/references (immutable)

- *uses_eggs* now takes a value of type &Eggs, which is "immutable reference to Eggs."
- Inside *uses_eggs* , we don't need to explicitly dereference the eggs value, this is done automatically by Rust.
- In main, we can now use x in two calls to *uses_eggs* .
- In order to create a reference from a value, we use & in front of the variable.

Question: What is the output of the program?

Borrows/references (immutable)

The output from this program is:

```
Before uses_eggs
I consumed : Eggs(1)
I consumed : Eggs(1)
After uses_eggs
Dropping Eggs: Eggs(1)
```

Multiple live references

We can allow two references to x to live at the same time.

```
fn main() {
    let x: Eggs = Eggs(1);
    let y: &Eggs = &x;
    println!("Before uses_eggs");
    uses_eggs(&x);
    uses_eggs(y);
    println!("After uses_eggs");
}
```

This is allowed because:

- Multiple read-only references cannot result in data races
- The lifetime of the value outlives the references to it.

Reference outlives value

What about this program?

```
fn main() {
    let x: Eggs = Eggs(1);
    let y: &Eggs = &x;
    println!("Before uses_eggs");
    uses_eggs(&x);
    std::mem::drop(x); //Delete x
    uses_eggs(y);
    println!("After uses_eggs");
}
```

This results in the error message:

```
error[E0505]: cannot move out of `x` because it is borrowed
```

Mutable reference with other references

- In order to avoid data races, Rust does not allow value to be referenced mutably and accessed in any other way at the same time.
- Rust tracks mutability at the type level.

```
fn main() {
    let mut x: Eggs = Eggs(1);
    let y: &mut Eggs = &mut x;
    println!("Before uses_eggs");
    uses_eggs(&x); //will gives an error
    uses_eggs(y);
    println!("After uses_eggs");
}
```

Default

- By default, variables are immutable.

```
#[derive(Debug)]
struct Eggs(i32);

fn main() {
    let mut x = Eggs(1);

    x.0 = 2; // changes the 0th value inside the egg

    println!("{:?}", x);
}
```

Question: What will happen if we remove mut?

Removing mutable

- As mentioned before, the following code will not compile:

```
#[derive(Debug)]
struct Eggs(i32);

fn main() {
    let x = Eggs(1);

    x.0 = 2; // changes the 0th value inside the egg

    println!("{:?}", x);
}
```

```
 --> src\main.rs:7:5
  |
5 |     let x = Eggs(1);
  |         - help: consider changing this to be mutable: `mut
x`
6 |
7 |     x.0 = 2; // changes the 0th value inside the egg
  |     ^^^^^^^ cannot assign
```

Adding mutable back

```
#[derive(Debug)]
struct Eggs(i32);

fn main() {
    let mut x = Eggs(1);

    x.0 = 2; // changes the 0th value inside the egg

    println!("{:?}", x);
}
```

```
Eggs(2)
```

Moving into mutable

- What about this code with the *egg* function?

```
#[derive(Debug)]
struct Eggs(i32);

fn main() {
    let x = Eggs(1);
    egg(x);
}

fn egg(mut x: Eggs) {

    x.0 = 2; // changes the 0th value inside the egg

    println!("{:?}", x);
}
```

Moving into mutable

- The code runs:

```
Eggs(2)
```