Lecture 10

Lifetimes in Rust

Problem One: Storing a borrow in a struct

```
struct Object {
    number: u32
}

struct Multiplier {
    object: &Object,
    mult: u32
}

fn print_borrower_number(mu: Multiplier) {
    println!("Result: {}", mu.object.number * mu.mult);
}

fn main() {
    let obj = Object { number: 5 };
    let obj_times_3 = Multiplier { object: &obj, mult: 3
};
    print_borrower_number(obj_times_3);
}
```

Problem One: Storing a borrow in a struct

- Multiplier is a structure that provides a way to multiply a number without having to touch the original object at all.
- This code will fail to compile, spitting out some weird lifetime error.

```
multiplier.rs:6:13: 6:20 error: missing lifetime
specifier [E0106]
multiplier.rs:6     object: &Object,
                            ^~~~~~~
multiplier.rs:6:13: 6:20 help: run `rustc --explain
E0106` to see a detailed explanation
```

Problem Two: Borrowed types in a function

- Let's use the power of borrowing to write another handy function: this time, to add an Object to another mutable Object, without having to transfer ownership.

```
fn object_combinator(a: &mut Object, b: &Object) -> &mut
Object {
    a.number = a.number + b.number;
    a
}

fn main() {
   let mut a = Object { number: 3 };
   let b = Object { number: 4 };
   println!("Result: {}", object_combinator(&mut a,
&b).number);
}
```

Problem Two: Borrowed types in a function

- This gives us the same error as above. What the hell, Rust?!

```
multiplier.rs:10:53: 10:64 error: missing lifetime
specifier [E0106]
multiplier.rs:10 fn object_combinator(a: &mut Object, b:
&Object) -> &mut Object {

^~~~~~~~~~~
multiplier.rs:10:53: 10:64 help: run `rustc --explain
E0106` to see a detailed explanation
```

Birth of a Lifetime

- Any borrow must last for a scope no greater than that of the owner.
- You may have either 1+ immutable borrow(s) or exactly 1 mutable borrow at a time - never both.
- How is the compiler supposed to check if we're abiding by these rules? Lifetimes are the answer.
- Lifetimes are used to describe how long objects live for.

Birth of a Lifetime

- When you make a new variable, the compiler attaches a lifetime to it. Then, by comparing all of the lifetimes to each other, the compiler can know whether your code is correct.

Rust enforces [the borrowing rules] through lifetimes. Lifetimes are effectively just names for scopes somewhere in the program. Each reference, and anything that contains a reference, is tagged with a lifetime specifying the scope it's valid for.

* Rustonomicon

Birth of a Lifetime

- Variables go away when your code leaves their scope in the opposite order that they were created.

```
struct RefObject<'x>(&'x u32);
fn steal_a_var<'x>(o: RefObject<'x>) {
    println!("{}", o.0);
}

fn main() {
    let a = 3; // a is created in main()s scope
    let mut b = &a; // b is created in main()s scope
    let c = RefObject(&b); // c is created in main()s scope
    steal_a_var(c); // c is moved out of main()s scope.
    // c now lives as long as steal_a_var()s scope.
    // steal_a_var()s scope ends so c goes away
    // d is created in main()s scope
    let d = &mut b;
}
```

Birth of a Lifetime

- When main()s scope ends, it kills all the variables inside it So…
  - d goes away, as it was declared last
  - b goes away, as it was declared second-last
  - a goes away, as it was declared third-last
- When a variable is created, it creates its own little mini-scope that lasts for as long as it is needed lasts until the scope containing it ends.
- For example, d goes away first because its mini-scope ends before b’s does.
- Let me rewrite this example, making the scopes more obvious:

Birth of a Lifetime

```
fn main() {
    // The `'a: {}` syntax used here isn't actually valid,
    // we are using it to show you the scope(s)
    'a: {
        let a = 3;
        'b: {
            let mut b = &a;
            'c: {
                let c = RefObject(&b);
                steal_a_var(c);
            } // c goes away
            'd: {
                let d = &mut b;
            } // d goes away
        } // b goes away
    } // a goes away
}
```

Fixing Our Problems

- Rust now looks at all the lifetimes:
- it sees that 'a is bigger than 'c and 'd, and knows that there are no borrows lasting longer than what they borrow.
- it sees that 'c and 'd are separate from each other and knows that the rules regarding mutable borrows are not broken.
- Now we at least know what lifetimes are, let’s use them to solve the two problems.

Problem One: Storing a borrow in a struct

```
struct Object {
    number: u32
}

struct Multiplier {
    object: &Object,
    mult: u32
}
```

- Rust is complaining cannot figure your lifetimes out for you.
- Object could live for any odd lifetime - possibly one smaller than Multiplier’s!

- Rust wants us to constrain the Multiplier so that it can check your code.

Problem One: Storing a borrow in a struct

- Here is the solution:

```
struct Multiplier<'a> {
    object: &'a Object,
    mult: u32
}
```

- Now, this code says "I have an object here called Multiplier that lives as long as the lifetime 'a. Given that lifetime, I have an object inside me that lasts as least as long as 'a."
- This then "links" the lifetime of a Multiplier to the lifetime of the Object inside it, making sure the Object won't go away before Multiplier does.

Problem Two: Borrowed types in a function

```
fn object_combinator(a: &mut Object, b: &Object) -> &mut Object
{
    a.number = a.number + b.number;
    a
}

fn main() {
   let mut a = Object { number: 3 };
   let b = Object { number: 4 };
   println!("Result: {}", object_combinator(&mut a,&b).number);
}
```

- The problem is that, we are giving object_combinator two borrows with possibly different lifetimes, and returning one borrow.
- How is Rust supposed to figure out how long the returned value lives for?

Problem Two: Borrowed types in a function

- object_combinator adds the second Object (b)'s number to a, and then returns a. Therefore, it's a's lifetime we care about here, as that's what we're returning.
- Let's annotate our function with lifetimes.

```
fn object_combinator<'a, 'b>(a: &'a mut Object, b: &'b Object)
-> &'a mut Object {
    a.number = a.number + b.number;
    a
}
```

- This says "I have an object_combinator which takes in a borrow to an Object that lives for lifetime 'a, and one that lives as long as the lifetime 'b.
- I'm going to return a borrow to an object that lives as long as the lifetime 'a."

‘static

- The 'static lifetime is a special lifetime which means “lives for the entire program”.

```
// You’ll probably see it when dealing with strings:
let string: &'static str = "I'm a string! Yay!";
// and static things:
static UNIVERSE_ANSWER: u32 = 42;
let answer_borrow: &'static u32 = &UNIVERSE_ANSWER;
```

Lifetimes --Basics

- Lifetimes coming into a function are called *input lifetimes*.
- Lifetimes returned from functions are *output lifetimes*.
- If there is one input lifetime, it is assigned to all output lifetimes. (Example: fn foo<'a>(bar: &'a str) -> &'a str)
- If there are many input lifetimes, but one of them is a reference to self, the lifetime of self is assigned to all output lifetimes.
- For more about lifetimes:

  https://doc.rust-lang.org/nomicon/lifetime-elision.html

Problem

```
#[derive(Debug)]
struct Earth {
  location: String,
}
#[derive(Debug)]
struct Dinosaur {
  location: &Earth,
  name: String,
}
fn main() {
  let new_york = Earth {location: "New York, NY".to_string(),};
  let t_rex = Dinosaur { location: &new_york, name: "T
Rex".to_string(),
  };
  println!("{:?}", t_rex);
}
```

```
error[E0106]: missing lifetime specifier
 --> src\main.rs:7:13
  |
7 |   location: &Earth,
  |             ^ expected lifetime parameter

error: aborting due to previous error
```

Solution

```
#[derive(Debug)]
struct Earth {
  location: String,
}
#[derive(Debug)]
struct Dinosaur<'a> {
  location: &'a Earth,
  name: String,
}
fn main() {
  let new_york = Earth {location: "New York, NY".to_string(),};
  let t_rex = Dinosaur { location: &new_york, name: "T
Rex".to_string(),
  };
  println!("{:?}", t_rex);
}
```

Problem

```
fn message_and_return(msg: &String, ret: &String) -> &String {
    println!("Printing the message: {}", msg);
    ret
}

fn main() {
    let name = String::from("Alice");
    let msg = String::from("This is the message");
    let ret = message_and_return(&msg, &name);
    println!("Return value: {}", ret);
}
```

```
error[E0106]: missing lifetime specifier
 --> src\main.rs:1:54
  |
1 | fn message_and_return(msg: &String, ret: &String) -> &String {
  |                                                      ^ expected lifetime parameter
  |
  = help: this function's return type contains a borrowed value, but the signature does no
t say whether it is borrowed from `msg` or `ret`
```

Solution

```
fn message_and_return<'a, 'b>(msg: &'a String, ret: &'b String)
-> &'b String {
    println!("Printing the message: {}", msg);
    ret
}
```

Problem

```
struct Person {
    name: String,
    age: u32,
}
fn get_older_name(person1: &Person, person2: &Person) ->&String
{
    if person1.age >= person2.age {&person1.name}
    else {&person2.name}
}
fn main() {
    let alice = Person {name: String::from("Alice"), age: 30,};
    let bob = Person {name: String::from("Bob"), age: 35,};
    let name = get_older_name(&alice, &bob);
    println!("Older person: {}", name);
}
```

```
error[E0106]: missing lifetime specifier
 --> src\main.rs:6:58
  |
6 | fn get_older_name(person1: &Person, person2: &Person) -> &String {
  |                                                          ^ expected lifetime parameter
  |
  = help: this function's return type contains a borrowed value, but the signature does no
t say whether it is borrowed from `person1` or `person2`
```

Solution 1: Doesn't Work!

```
struct Person {
    name: String,
    age: u32,
}
fn get_older_name<'a, 'b>(person1: &'a Person, person2: &'b
Person) -> &String {
    if person1.age >= person2.age {&person1.name}
    else {&person2.name}
}
fn main() {
    let alice = Person {name: String::from("Alice"), age: 30,};
    let bob = Person {name: String::from("Bob"), age: 35,};
    let name = get_older_name(&alice, &bob);
    println!("Older person: {}", name);}
```

```
error[E0106]: missing lifetime specifier
 --> src\main.rs:6:72
  |
6 | fn get_older_name<'a, 'b>(person1: &'a Person, person2: &'b Person) -> &String
  |                                                                        ^ expected life
time parameter
  |
  = help: this function's return type contains a borrowed value, but the signature does no
t say whether it is borrowed from `person1` or `person2`
```

Solution 2:

```
struct Person {
    name: String,
    age: u32,
}
fn get_older_name<'a>(person1: &'a Person, person2: &'a Person)
-> &'a String {
    if person1.age >= person2.age {&person1.name}
    else {&person2.name}
}
fn main() {
    let alice = Person {name: String::from("Alice"), age: 30,};
    let bob = Person {name: String::from("Bob"), age: 35,};
    let name = get_older_name(&alice, &bob);
    println!("Older person: {}", name);
}
```

```
Older person: Bob
```