CIS 1905

Lecture 05: Lifetimes

Lifetimes are cool

Logistics

Homeworks

- HW2 Extended to Thursday 11:59pm
- All assignments are now due on Monday 11:59pm, instead of Sunday 11:59pm

Final project proposal due March 12th

- Rubric released by Sunday night

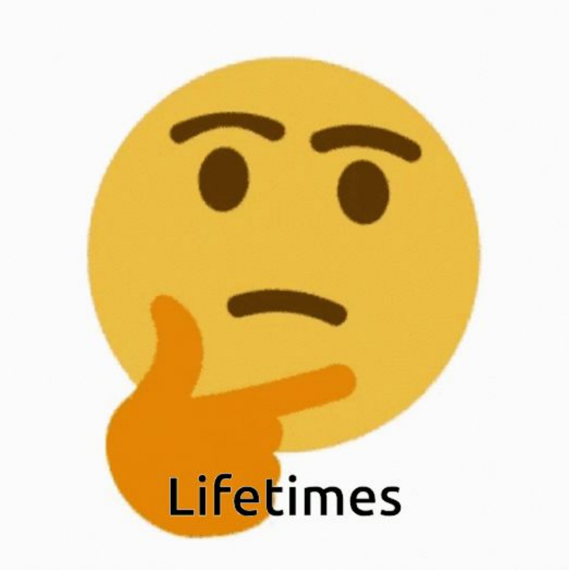
PollEv: Questions?

pollev.com/cis1905rust915

Zero Cost Abstraction

‼️ Warning ‼️

Use your brain, not the compiler suggestions (but also read the compiler suggestions)



Memory Safety at Zero Cost

1. **Ownership:** Each object should have exactly one owner; memory is dropped when the owner leaves scope
2. **Borrowing:** When you "borrow" ownership, you get **exclusively either**:
   a. read-only access (any number of borrows)
   b. read-write access (exactly one borrow)
3. Today: Lifetimes

What about invalid references?

```
let a = /* some value */
let b = &a
/* a dies */
```

What is b now?

Lifetime

How long a particular reference is valid

Every reference has a *lifetime*

Lifetime: the scope for which a particular reference is valid

- Input lifetimes
  - Function/method parameters
  - Struct definition
- Output lifetimes
  - Return values
  - Impl for struct

```
pub fn nested_lifetimes() {
    let s = "hello";
    // s lifetime: 'static
    {
        let a = String::from("x");
        // a lifetime: 'lifetime1
        {
            let b = String::from("y");
            // b lifetime: 'liftime2
            // where we have an implicit ordering:
            // 'static >= 'lifetime1 >= 'lifetime2

            // Here s, a, and b are in scope
            println!("{}", constant_str_dummyargs(&a, &b));
            println!("{}", constant_str_dummyargs(&a, s));
            println!("{}", constant_str_dummyargs(s, &b));
        }

        println!("{}", constant_str_dummyargs(&a, s));
        // Here s and a are in scope
    }

    // Here only s is in scope.
    println!("{}", constant_str_dummyargs(s, s));
}
```

Lifetime Annotations

Lifetime Annotations

Sometimes, the compiler need a little help

```
fn select(x: &str, y: &str, condition: bool) -> &str {
    if condition {
        x
    } else {
        y
    }
}
```

Lifetime Annotations

Sometimes, the compiler need a little help

```
fn select<'a>(x: &'a str, y: &'a str, condition: bool) -> &'a str {
    if condition {
        x
    } else {
        y
    }
}
```

Lifetime Annotations

A notation that describes relative lifetimes of references

`x: &'a i32` x lives for `‘a` long

What does that mean?

`’a` helps put the “lifetime” of `x` in perspective of other variables:

```
struct Foo<'a> {
    x: &'a i32,
}
fn skip_prefix<'a, 'b>(line: &'a str, prefix: &'b
str) -> &'a str {
    // ...
}
```

Lifetime Annotations

Lifetime Parameter,
similar to Generic<T>

Output Parameter(s)

```
fn skip_prefix<'a, 'b>(line: &'a str, prefix: &'b str) -> &'a str {
    // ...
}
```

Input Parameter(s)

Lifetime Annotations

```
struct Foo<'n> {
    x: &'n i32,
}
fn skip_prefix<'a, 'b>(line: &'a str, prefix: &'b str) -> &'a str {
    // ...
}
fn<’hi> foo(param1: &'hi type) -> &'hi return_value
```

Evaluating lifetime annotations

Lifetime in which all related references are valid

- When conflict: smallest lifetime of all references

```
fn main() {
    let string1 = String::from("Hello");
    let result;
    {
        let string2 = String::from("World");
        result = longest(&string1, &string2);
    } // string2 goes out of scope here
    println!("The longest string is {}", result);
}
fn longest<'a>(x: &'a str, y: &'a str) -> &'a str {
    if x.len() > y.len() { x } else { y }
}
```

Evaluating lifetime annotations

```
error[E0597]: `string2` does not live long
enough
  --> src/main.rs:35:36
   |
34 |         let string2 =
String::from("World");
   |             ------- binding `string2`
declared here
35 |         result = longest(&string1,
&string2);
   |
^^^^^^^^ borrowed value does not live long
enough
36 |     } // string2 goes out of scope here
   |     - `string2` dropped here while still
borrowed
37 |     println!("The longest string is {}",
result);
   |
------ borrow later used here
```

```rust
fn main() {
    let string1 = String::from("Hello");
    let result;
    {
        let string2 = String::from("World");
        result = longest(&string1, &string2);
    } // string2 goes out of scope here
    println!("The longest string is {}", result);
}
fn longest<'a>(x: &'a str, y: &'a str) -> &'a str {
    if x.len() > y.len() { x } else { y }
}
```

Evaluating lifetime annotations

Lifetime in whic

- When conf

Fixed. All gud?

```
fn main() {
    let string1 = String::from("Hello");
    let result;
    {
        let string2 = String::from("World");
        result = longest(&string1, &string2);
        println!("The longest string is {}", result);
    }
}
fn longest<'a>(x: &'a str, y: &'a str) -> &'a str {
    if x.len() > y.len() { x } else { y }
}
```

Evaluating lifetime annotations

Lifetime in whic

- When conf

What is ‘a in the call in main()?

```
fn main() {
    let string1 = String::from("Hello");
    let result;
    {
        let string2 = String::from("World");
        result = longest(&string1, &string2);
        println!("The longest string is {}", result);
    }
}
fn longest<'a>(x: &'a str, y: &'a str) -> &'a str {
    if x.len() > y.len() { x } else { y }
}
```

‘static lifetime

Lifetime that ***can*** extend until the end of the program

```
let x: &'static str = "Hello, world.";
```

Wait a second

```
pub fn nested_lifetimes() {
    let s = "hello";
    // s lifetime: 'static
    {
        let a = String::from("x");
        // a lifetime: 'lifetime1
        {
            let b = String::from("y");
            // b lifetime: 'liftime2
            // where we have an implicit ordering:
            // 'static >= 'lifetime1 >= 'lifetime2

            // Here s, a, and b are in scope
            println!("{}", constant_str_dummyargs(&a, &b));
            println!("{}", constant_str_dummyargs(&a, s));
            println!("{}", constant_str_dummyargs(s, &b));
        }

        println!("{}", constant_str_dummyargs(&a, s));
        // Here s and a are in scope
    }

    // Here only s is in scope.
    println!("{}", constant_str_dummyargs(s, s));
}
```

Lifetime Annotations

Lifetime annotations do not **change** the actual lifetime, it constrains it:

```rust
fn extend_lifetime<'a>() -> &'a i32 {
    let x = 42;
    &x // Return a reference to x with a lifetime 'a
}

fn main() {
    let y = extend_lifetime(); // does not live long enough
    println!("{}", y);
}
```

Lifetime Ellison

Lifetime Ellison

Usually, you don’t need to annotate lifetimes, because the compiler can infer it :D

```
fn print(s: &str); // elided
fn print<'a>(s: &'a str); // expanded

fn debug(lvl: u32, s: &str); // elided
fn debug<'a>(lvl: u32, s: &'a str); // expanded
```

Lifetime Ellison Rules

1. Each elided lifetime in a function’s arguments becomes a distinct lifetime parameter.

```
fn my_func<’a, ‘b>(x: &‘a str, y: &‘b str);
```

2. If there is exactly one input lifetime, elided or not, that lifetime is assigned to all elided lifetimes in the return values of that function.

```
fn chop<’a>(x: &‘a str) -> (&‘a str, &‘a str);
```

3. If there are multiple input lifetimes, but one of them is `&self` or `&mut self`, the lifetime of self is assigned to all elided output lifetimes.

```
fn split<’a, ‘b>(&’a self, delimiter: &’b str) -> &’a
str;
```

Anonymous Lifetime (’_)

Up to the compiler to resolve the lifetime

- When used in **argument position,** ‘_ gets turned into an arbitrary unique lifetime
- When used in **output position,** ’_ is type inferred

```
fn foo(x: &str, y: &'_ str) -> &'_ str {}
```

What is &’_ and what is &’_?

Doesn’t the compile already do this with lifetime ellison?

Explicit elided lifetimes or anonymous lifetimes

They are used when you need to specify a lifetime due to syntax requirements but still want the compiler to infer the lifetime. They're a way to tell the compiler that the specific lifetime isn't important for understanding the code and can be inferred

```
fn parse_input(s: &str) -> impl Iterator<Item=Foo> + '_

fn foo<'a>(x: cell::Ref<'a, Foo>) -> usize
```

PollEv

pollev.com/cis1905rust915

Playground

http://tinyurl.com/rustpg