SPRING 2024

INTRO TO RUST LANG

LIFETIMES

Benjamin Owad, David Rudo, and Connor Tsui

Today: Lifetimes

We've used the term "lifetime" a few times before, and today we're going to explore what exactly it means.

- What is `'a` lifetime?
- How to think about lifetimes
- Other perspectives...

Lifetimes

Lifetimes are all about references, and **nothing** else.

- Informal definition:
  Lifetimes provide a way for Rust to validate pointers at compile time
- Formal definition:
  Lifetimes are named regions of code that a reference must be valid for
- *Remember that references are just pointers with constraints!*

Lifetimes vs Generics and Traits

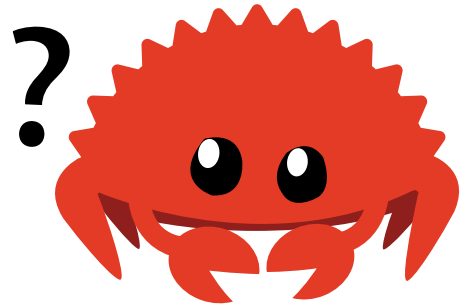
Lifetimes are similar to trait bounds.

- Traits ensure that a generic type has the behavior we want
- Lifetimes ensure that references are valid for as long as we need them to be

Validating References

The main goal of lifetimes it to prevent *dangling references*.

```
fn main() {
    let r;

    {
        let x = 5;
        r = &x;
    }

    println!("r: {}", r);
}
```



- What is the issue with this code?

Validating References

```
error[E0597]: `x` does not live long enough
 --> src/main.rs:6:13
  |
6 |         r = &x;
  |             ^^ borrowed value does not live long enough
7 |     }
  |     - `x` dropped here while still borrowed
8 |
9 |     println!("r: {}", r);
  |                       - borrow later used here
```

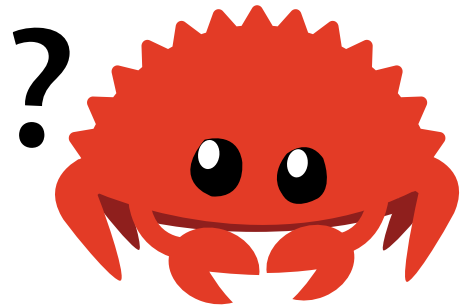
- The value that `r` refers to has gone out of scope before we could use it
- The scope of `r` is "larger" than the scope of `x`

The Borrow Checker

The Rust compiler's borrow checker will compare scopes to determine whether all borrows are valid.

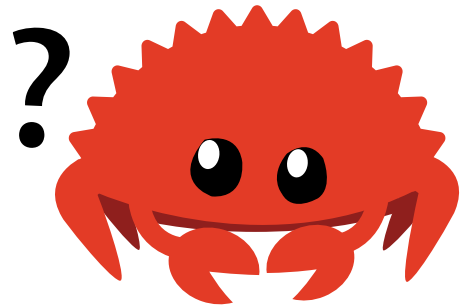
Here is the same code, but with a lifetime diagram:

```
fn main() {
    let r;                // ---------+-- 'a
                          //          |
    {                     //          |
        let x = 5;        // -+-- 'b  |
        r = &x;           //  |       |
    }                     // -+       |
                          //          |
    println!("r: {}", r); //          |
}                         // ---------+
```



The Borrow Checker

```
fn main() {
    let r;                  // ----------+-- 'a
                            //           |
    {                       //           |
        let x = 5;          // -+-- 'b   |
        r = &x;             //  |        |
    }                       // -+        |
                            //           |
    println!("r: {}", r); //           |
}                           // ----------+
```

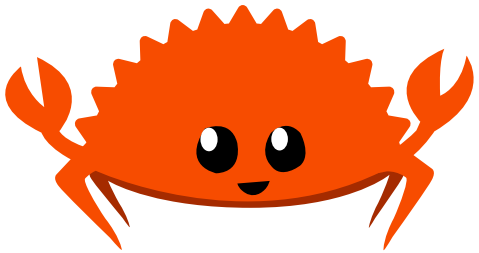


- The borrow checker will compare the "size" of the two lifetimes
  - `r` has a lifetime of `'a`
  - `r` refers to a variable with lifetime `'b`
  - Rejects because `'b` is shorter than `'a`

Placating the Borrow Checker

We can fix this code by removing the scope.

```
fn main() {
    let x = 5;                  // ----------+-- 'b
                                //           |
    let r = &x;                 // --+-- 'a  |
                                //   |       |
    println!("r: {}", r);       //   |       |
                                // --+       |
}                               // ----------+
```



- `x` "outlives" `r`, so `r` can reference `x`

Generic Lifetimes

Let's try to write some string functions.

```
fn main() {
    let string1 = String::from("abcd");
    let string2 = "xyz";

    let result = longest(string1.as_str(), string2);
    println!("The longest string is {}", result);
}
```

We want this output:

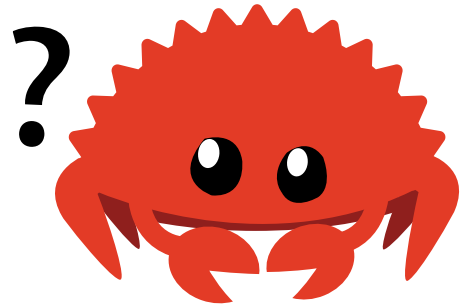
```
The longest string is abcd
```

- Let's implement `longest`!

`longest`

Here's a first attempt:

```
fn longest(x: &str, y: &str) -> &str {
    if x.len() > y.len() {
        x
    } else {
        y
    }
}
```



- *We don't want to take ownership, so we take* `&str` *inputs*

`longest` error

Unfortunately, our attempt will not compile:

```
error[E0106]: missing lifetime specifier
 --> src/main.rs:9:33
  |
9 | fn longest(x: &str, y: &str) -> &str {
  |               ----     ----     ^ expected named lifetime parameter
  |
  = help: this function's return type contains a borrowed value,
    but the signature does not say whether it is borrowed from `x` or `y`
help: consider introducing a named lifetime parameter
  |
9 | fn longest<'a>(x: &'a str, y: &'a str) -> &'a str {
  |           ++++     ++          ++           ++
```

`longest` error

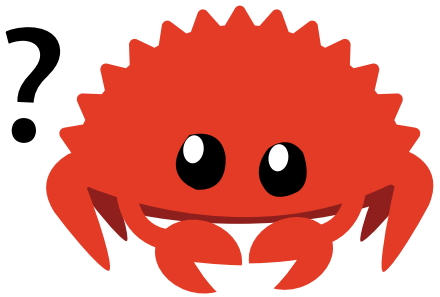
The help text from the compiler error reveals some useful information:

```
  = help: this function's return type contains a borrowed value,
    but the signature does not say whether it is borrowed from `x` or `y`
```

- Rust can't figure out if the reference returned refers to `x` or `y`
- In fact, neither do we!

`longest` error

```
fn longest(x: &str, y: &str) -> &str {
    if x.len() > y.len() {
        x
    } else {
        y
    }
}
```



- We don't know which execution path this code will take
- We don't know the lifetimes of the input references either
- Thus we cannot determine the lifetime we return!
- We will need to *annotate* these references

Lifetime Annotation Syntax

We can annotate lifetimes with generic parameters that start with a `'`, like `'a`.

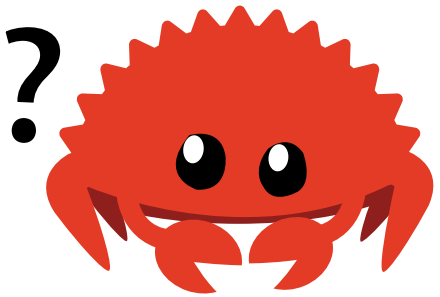
```
&i32          // a reference
&'a i32       // a reference with an explicit annotated lifetime
&'a mut i32   // a mutable reference with an explicit lifetime

&'hello usize // annotations can be any word or character,
&'world bool  // as long as it starts with a tick (')
```

- Annotations do not change the how long references live, they only describe the relationship between lifetimes of references
- One annotation by itself has little meaning

`longest` Lifetimes

Let's return back to our `longest` function.

```
fn longest(x: &str, y: &str) -> &str {
    if x.len() > y.len() {
        x
    } else {
        y
    }
}
```



- What do we want the function signature to express?
- What should the relationship be between the lifetimes of the references?

`longest` Lifetimes

What exactly are we returning?

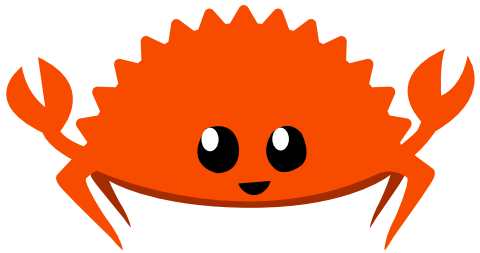
```
if x.len() > y.len() {
    x
} else {
    y
}
```

- We return either `x` or `y`, which each have their own lifetimes
- We want the returned reference to be valid as long as *both* input references `x` and `y` are valid
- So we want lifetimes of `x` and `y` to *outlive* the returned lifetime

`longest` Lifetimes

Since lifetimes are a kind of generic parameter, we must declare them like normal generic type parameters.

```
fn longest<'a>(x: &'a str, y: &'a str) -> &'a str {
    if x.len() > y.len() {
        x
    } else {
        y
    }
}
```



- This will compile now!
- *Remember that these lifetime annotations don't change the lifetimes of any values*

Lifetime Annotations in Functions

We can extrapolate a lot from a function's signature, even without the body.

```
fn longest<'a>(x: &'a str, y: &'a str) -> &'a str;
```

- This function takes two string slices (`x` and `y`) that live at least as long as lifetime `'a`
- The string slice returned (the longer of `x` or `y`) will also live at least as long as `'a`

Lifetime Annotations in Functions

```
fn longest<'a>(x: &'a str, y: &'a str) -> &'a str;
```

- When calling `longest`, the lifetime that is substituted for `'a` is the intersection of the lifetimes of `x` and `y`
- In practice, this means the lifetime returned by `longest` is the same as the smaller of the two input lifetimes

Borrow Checker Example 1

Let's look at some examples where the borrow checker is and isn't happy.

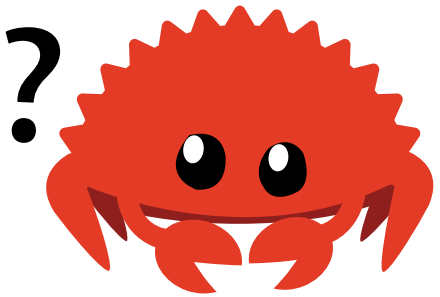
```
fn main() {
    let string1 = String::from("long string is long");

    {
        let string2 = String::from("xyz");
        let result = longest(string1.as_str(), string2.as_str());
        println!("The longest string is {}", result);
    }
}
```

- `string1` is valid in the outer scope
- `string2` is valid in the inner scope
- `result` should only be valid in the smaller scope (by our lifetime annotations)
  - Since `println!` is in the smaller (inner) scope, this works!

Borrow Checker Example 2

Let's reorder some things around.

```
fn main() {
    let string1 = String::from("xyz");
    let result;
    {
        let string2 = String::from("long string is long");
        result = longest(string1.as_str(), string2.as_str());
    }
    println!("The longest string is {}", result);
}
```



- `result` should only be valid in the smaller (inner) scope, but we try to reference it in the outer scope

Borrow Checker Example 2

Sure enough, this does not compile, and Rust gives us this error:

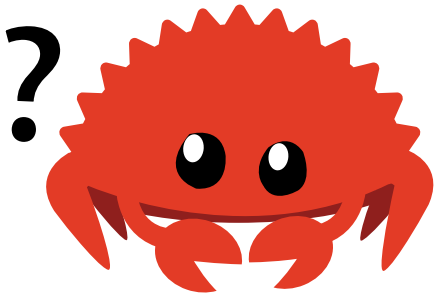
```
error[E0597]: `string2` does not live long enough
 --> src/main.rs:6:44
  |
6 |         result = longest(string1.as_str(), string2.as_str());
  |                                            ^^^^^^^^^^^^^^^^^
                       borrowed value does not live long enough
7 |     }
  |     - `string2` dropped here while still borrowed
8 |     println!("The longest string is {}", result);
  |                                          ------ borrow later used here
```

Borrow Checker Example 3

What if we knew (as the programmer) that `string1` is always longer than `string2`?

Let's switch the strings around:

```
let string1 = String::from("long string is long");
let result;
{
    let string2 = String::from("xyz");
    result = longest(string1.as_str(), string2.as_str());
}
println!("The longest string is {}", result);
```



Borrow Checker Example 3

```
let string1 = String::from("long string is long");
let result;
{
    let string2 = String::from("xyz");
    result = longest(string1.as_str(), string2.as_str());
}
println!("The longest string is {}", result);
```

- Even though we know (as a human) that the reference will be valid, the compiler does not know
- We even told the compiler that the returned lifetime would be the same as the smaller of the input lifetimes!

Avoiding Lifetime Annotations

Suppose we wanted to always return the first input, `x`.

```
fn first<'a>(x: &'a str, y: &str) -> &'a str {
    x
}
```

- We don't need to annotate `y` with `'a`, because the return value doesn't care about `y`'s lifetime

Lifetimes of Return Values

The lifetime of a return value *must* match the lifetime of one of the inputs.

```
fn dangling<'a>(x: &str, y: &str) -> &'a str {
    let result = String::from("really long string");
    result.as_str()
}
```

- If it didn't depend on an input, then it would *always* be a dangling reference!

Lifetime Elision

All references must have a lifetime. But we've seen many references without lifetime annotations...

This is a version of a function we saw back in week 2:

```
fn first_word(s: &str) -> &str {
    let bytes = s.as_bytes();

    for (i, &item) in bytes.iter().enumerate() {
        if item == b' ' {
            return &s[0..i];
        }
    }

    &s[..]
}
```

- There are no lifetime annotations here!

Story Time

Long ago, in the dark ages of the 2010s, every reference needed an explicit lifetime.

```
fn first_word<'a>(s: &'a str) -> &'a str {
```

- Before Rust 1.0, every single `&` needed an explicit `'something` annotation
- This became incredibly repetitive, and so the Rust team programmed the borrow checker to infer lifetime annotation patterns of certain situations
- These patterns are called the *lifetime elision rules*

Lifetime Elision

- Lifetime elision does not provide full inference, it will only infer when it is absolutely sure it is correct
- Lifetimes on function or method arguments are called *input lifetimes*, and lifetimes on return values are called *output lifetimes*
- There are only 3 lifetime elision rules, the first for input lifetimes, the last two for output lifetimes

Lifetime Elision Rule 1

The first rule is that the compiler will assign a different lifetime parameter for each input lifetime.

```
fn foo(x: &i32);
fn foo<'a>(x: &'a i32);

fn bar(x: &i32, y: &i32);
fn bar<'a, 'b>(x: &'a i32, y: &'b i32);
```

Lifetime Elision Rule 2

The second rule is that if there is only 1 input lifetime parameter, then it is assigned to all output lifetimes.

```
fn foo(x: &i32) -> &i32;
fn foo<'a>(x: &'a i32) -> &'a i32;

fn bar(arr: &[i32]) -> (&i32, &i32);
fn bar<'a>(arr: &'a [i32]) -> (&'a i32, &'a i32);
```

Lifetime Elision Rule 3

If there are multiple input lifetime parameters, but the first parameter is `&self` or `&mut self`, the lifetime of `&self` is assigned to all output lifetimes.

- This only applies to methods
- Makes writing methods much nicer!
- *Examples to come later...*

Lifetime Elision Example 1

Let's pretend we are the compiler, and let's attempt to apply the lifetime elision rules to `first_word`.

```
fn first_word(s: &str) -> &str;
```

Lifetime Elision Example 1

We apply the first rule, which specifies that each parameter gets its own lifetime.

```
fn first_word<'a>(s: &'a str) -> &str;
```

Lifetime Elision Example 1

The second rule specifies that the lifetime of the single input parameter gets assigned to all output lifetimes, so the signature becomes this:

```
fn first_word<'a>(s: &'a str) -> &'a str;
```

- Since all references have lifetime annotations, we're done!

Lifetime Elision Example 2

So why didn't elision work with `longest`? Let's trace it out!

We start with this signature without annotations:

```
fn longest(x: &str, y: &str) -> &str;
```

Lifetime Elision Example 2

Let's apply the first rule and get annotations for all inputs.

```
fn longest<'a, 'b>(x: &'a str, y: &'b str) -> &str;
```

- What now?

Lifetime Elision Example 2

```
fn longest<'a, 'b>(x: &'a str, y: &'b str) -> &str;
```

- The second rule doesn't apply here, because there is more than 1 input lifetime (`'a` and `'b`)
- Since Rust cannot figure out what to do, it gives a compiler error to the programmer so they can write the annotations themselves

Lifetimes in Structs

So far, all of the `struct`s we've looked at have held *owned* type fields.

If we want a `struct` to hold a reference, we need to annotate them.

```
struct ImportantExcerpt<'a> {
    part: &'a str,
}

fn main() {
    let novel = String::from("Call me Ishmael. Some years ago...");
    let first_sentence = novel.split('.').next().expect("Could not find a '.'");
    let i = ImportantExcerpt {
        part: first_sentence,
    };
}
```

Lifetimes in Structs

```
struct ImportantExcerpt<'a> {
    part: &'a str,
}
```

- As with generic data types, we declare the name of the generic lifetime parameter inside angle brackets
- This annotation means an instance of `ImportantExcerpt` can't outlive the reference it holds in its `part` field

Lifetimes in `impl` Blocks

Similarly, we need to annotate `impl` blocks with lifetime parameters.

```
impl<'a> ImportantExcerpt<'a> {
    fn level(&self) -> i32 {
        3
    }
}
```

Lifetimes in Methods

Here is an example where the third elision rule is applied:

```
impl<'a> ImportantExcerpt<'a> {
    fn announce_and_return_part(&self, announcement: &str) -> &str {
        println!("Attention please: {}", announcement);
        self.part
    }
}
```

- The first rule gives both `&self` and `announcement` their own lifetimes
- The third rule gives the return lifetime the lifetime of `&self`

Putting it all together...

Let's briefly look at the syntax of specifying generic type parameters, trait bounds, and lifetimes all in one function!

```
fn longest_with_an_announcement<'a, T>(x: &'a str, y: &'a str, ann: T) -> &'a str
where
    T: Display,
{
    println!("Announcement! {}", ann);
    if x.len() > y.len() {
        x
    } else {
        y
    }
}
```

Lifetime Bounds

Lifetimes can be bounds, just like traits.

```
#[derive(Debug)]
struct Ref<'a, T: 'a>(&'a T);
```

- `Ref` contains a reference, with a lifetime of `'a`, to a generic type `T`
- `T` is bounded such that any references *in* `T` must live at least as long as `'a`
- Additionally, the lifetime of `Ref` may not exceed `'a`

Lifetime Bounds

Here is a similar example, but with a function instead of a `struct`.

```
fn print_ref<'a, T>(t: &'a T)
where
    T: Debug + 'a,
{
    println!("print_ref(t) is {:?}", t);
}
```

- `T` must implement `Debug`, and all references *in* `T` must outlive `'a`
- Additionally, `'a` must outlive this function call

Lifetime Bounds

Putting the `Ref` and `print_ref` together:

```
#[derive(Debug)]
struct Ref<'a, T: 'a>(&'a T);

fn print_ref<'a, T>(t: &'a T)
where
    T: Debug + 'a,
{
    println!("print_ref(t) is {:?}", t);
}

fn main() {
    let x = vec![9, 8, 0, 0, 8];
    let ref_x = Ref(&x);
    print_ref(&ref_x);
    // Prints to stdout: print_ref(t) is Ref([9, 8, 0, 0, 8])
}
```

Lifetime-bounded Lifetimes

We can have lifetimes that are bounded by other lifetimes.

```
// Takes in a `&'a i32` and return a `&'b i32` as a result of coercion
fn choose_first<'a: 'b, 'b>(first: &'a i32, _: &'b i32) -> &'b i32 {
    first
}

fn main() {
    let first = 2; // Longer lifetime
    {
        let second = 3; // Shorter lifetime
        println!("{} is the first", choose_first(&first, &second));
    }
}
```

- `'a: 'b` reads as "lifetime `'a` outlives `'b`"

The `'static` Lifetime

There is a special lifetime called `'static`.

```
let s: &'static str = "I have a static lifetime";
```

- `'static` implies that the reference will live until the end of the program (it is valid until the program stops running)
- Here, `s` is stored in the program binary, so it will always be valid!

`'static` Error Messages

You may see suggestions to use the `'static` lifetime in error messages.

```
fn foo() -> &i32 {
    let x = 5;
    &x
}
```

```
help: consider using the `'static` lifetime, but this is uncommon unless you're
      returning a borrowed value from a `const` or a `static`
  |
2 | fn foo() -> &'static i32 {
  |              +++++++
```

- Before making a change, think about if your reference will *really* live until the end of the program
- You may actually be trying to create a dangling reference!

`'static` vs `static`

There are two common ways to make a variable with a `'static` lifetime.

1. Make a string literal with has type `&'static str`
2. Make a constant with the `static` declaration

`'static` vs `static` Example

```
static NUM: i32 = 42;
static NUM_REF: &'static i32 = &NUM;

fn main() {
    let msg: &'static str = "Hello World";
    println!("{msg} {NUM_REF}!");
}
```

```
Hello World 42!
```

`'static` Memory Leaks

There is a third way: we can create `'static` values by *leaking memory*.

```
fn random_vec() -> &'static [usize; 100] {
    let mut rng = rand::thread_rng();
    let mut boxed = Box::new([0; 100]);
    boxed.try_fill(&mut rng).unwrap();
    Box::leak(boxed)
}

fn main() {
    let first: &'static [usize; 100] = random_vec();
    let second: &'static [usize; 100] = random_vec();
    assert_ne!(first, second)
}
```

- This allows us to *dynamically* create a `'static` reference!

The `'static` Bound

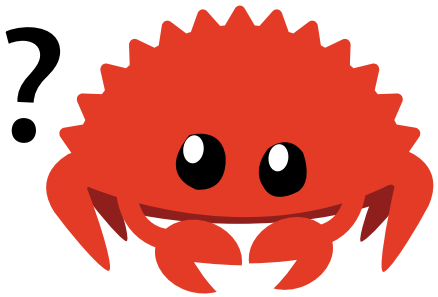
`'static` can also be used as a type bound. However...

- There is a subtle difference between the `'static` lifetime and the `'static` bound
- The `'static` bound means that the type does not contain any non-static references
- This means that all owned data implicitly has a `'static` bound, since owned data holds no references

`'static` Bound Example

Here's an example of using a `'static` bound.

```
fn print_it(input: impl Debug + 'static) {
    println!("'static value passed in is: {:?}", input);
}

fn main() {
    // i is owned and contains no references,
    // thus it has a 'static bound
    let i = 5;
    print_it(i);

    // oops, &i only has the lifetime defined by
    // the scope of main, so it's not 'static
    print_it(&i);
}
```



`'static` Bound Example

We get a compiler error:

```
error[E0597]: `i` does not live long enough
  --> src/lib.rs:15:15
   |
15 |     print_it(&i);
   |     ---------^^--
   |     |         |
   |     |         borrowed value does not live long enough
   |     argument requires that `i` is borrowed for `'static`
16 | }
   | - `i` dropped here while still borrowed
```

Review

- Rust has lifetimes to prevent dangling references
- The borrow checker will ensure that lifetimes are always valid
- Rust will allow you elide lifetime annotations in some situations

Further Reading

- You can find some more examples here: Rust By Example
- If you want to go *really* in depth, read the Rustonomicon chapter on lifetimes

Another Perspective

What is 'a lifetime?

- This is a great video made by `leddoo` that explains another way to think about lifetimes!
- Instead of lifetimes as regions of code or scopes, what if we thought about lifetimes as regions of memory?
- Let's watch it together!

Watch Party

What is 'a lifetime?

What is `'a` lifetime?

Some quick points:

- Thinking about lifetimes as regions of code can be confusing
- Instead, think about lifetimes as regions of valid memory
- Both interpretations are valid!

Feedback

If you have 5 minutes, please fill out the feedback form (on Piazza).

- It will help us make this semester better for you
- It will also help make future offerings of this course better for others!
- Feedback is anonymous, so please be honest

Homework 8

Homework 8 is a quiz on Gradescope!

- Think of homework 8 as a take-home midterm that only grades participation
- All 15 questions come from the experimental Brown Rust Book
  - They are doing active research in the best methods to teach Rust!
- *Please don't spend more than 1 hour on this*

Next Lecture: `Box` and Trait Objects

- Thanks for coming!