Rust Bootcamp 2021

Day 7

Day 7 Activities

- ✓ Lifetimes
- ✓ Iterators
- ✓ Overloading Operators
- ✓ Exercises

Lifetimes

Besides ownership, the other major compiler error in Rust has to do with the lifetime of references.

A *lifetime* is the scope for which a reference is valid. This prevents dangling references, or other pointer errors from languages like C.

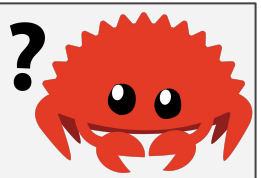
Most of the time lifetimes are implicit, and the compiler can figure them out. Sometimes, it may be necessary to explicitly annotate lifetime relationships.

https://doc.rust-lang.org/book/ch10-03-lifetime-syntax.html

Implicit Lifetimes

The Rust *borrow checker* is the main tool that validates lifetimes.

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

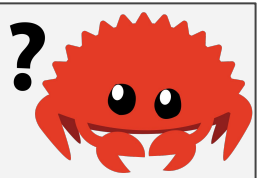


https://doc.rust-lang.org/book/ch10-03-lifetime-syntax.html#the-borrow-checker

Function Lifetimes

When returning a reference, the borrow checker will often complain.

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



The issue here is that the compiler can't figure out if the return time should use the lifetime from x or y.

https://doc.rust-lang.org/book/ch10-03-lifetime-syntax.html#generic-lifetimes-in-functions

Function Lifetimes

We can add lifetime annotation to the function to fix this:

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

Now the compiler knows that the lifetime for x and y is the same, as is the lifetime for the return value.

https://doc.rust-lang.org/book/ch10-03-lifetime-syntax.html#generic-lifetimes-in-functions

Lifetime Annotation Syntax

As seen on the previous slide, lifetimes are denoted with a single quote:

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

A single lifetime annotation doesn't have much meaning, as it is designed to show relationships between things. They are mostly used for functions and types, to clarify relationships between references.

https://doc.rust-lang.org/book/ch10-03-lifetime-syntax.html#lifetime-annotation-syntax

Lifetime Elision

While you can add lifetimes everywhere, this is not required.

The reason is historical: lifetimes used to be required everywhere, until the Rust team added several common patterns into the compiler so the borrow checker could infer those lifetimes. These are called the lifetime elision rules.

This is relevant, as additional patterns may be added in the future.

This is also how the compiler can suggest lifetime fixes, as common options have been programmed in when there is ambiguity.

https://doc.rust-lang.org/book/ch10-03-lifetime-syntax.html#lifetime-elision

Lifetime Annotation in Structs

Here we show how to store a reference inside a struct, using a lifetime.

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

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

The lifetime says that the struct itself is valid only as long as the reference it stores.

https://doc.rust-lang.org/book/ch10-03-lifetime-syntax.html#lifetime-annotations-i

Lifetime Annotation in Struct Methods

We can take our struct, and add a method to it:

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

Note that we are only required to specify the lifetime in the impl declaration.

Rust can automatically infer the lifetime for self and the return type.

https://doc.rust-lang.org/book/ch10-03-lifetime-syntax.html#lifetime-annotations-in-method-definitions

Static Lifetime

If a reference lives for the entire program, even past the scope of main, it has a ‘static lifetime.

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

All string literals have a ‘static lifetime because they are stored in the binary.

Remembering back to threads, if you tried using a reference across a thread boundary you probably got an error suggesting a ‘static lifetime. While you might be able to do that for string literals, most of the time it means you’re attempting to create a dangling reference, and need to instead move ownership.

https://doc.rust-lang.org/book/ch10-03-lifetime-syntax.html#the-static-lifetime

Iterators

In Rust, iterators are lazy - they have no effect until you call them.

```
let v1 = vec![1, 2, 3];

let v1_iter = v1.iter();
```

This code doesn't actually iterate the vector, just creates an iterator that is ready to do so. We can also get a mutable iterator:

```
let v1 = vec![1, 2, 3];

let v1_iter = v1.iter_mut();
```

https://doc.rust-lang.org/book/ch13-02-iterators.html

Looping on an Iterator

To actually do something with an iterator, we need to use it.

```
let v1 = vec![1, 2, 3];

let v1_iter = v1.iter();

for val in v1_iter {
    println!("Got: {}", val);
}
```

This gives us a reference *val* that references each element in the vector inside the loop.

https://doc.rust-lang.org/book/ch13-02-iterators.html

Iterator references without looping

We can also get references directly from the iterator with *next*:

```
let v1 = vec![1, 2, 3];

let mut v1_iter = v1.iter();

assert_eq!(v1_iter.next(), Some(&1));
assert_eq!(v1_iter.next(), Some(&2));
assert_eq!(v1_iter.next(), Some(&3));
assert_eq!(v1_iter.next(), None);
```

next returns an Optional, since it's not guaranteed there is another value. When the iterator runs out, it always returns None.

https://doc.rust-lang.org/book/ch13-02-iterators.html#the-iterator-trait-and-the-next-method

Iterator map

One useful function is *map*, which calls a function or closure on each element.

```
let v1: Vec<i32> = vec![1, 2, 3];

let v2: Vec<_> = v1.iter().map(|x| x + 1).collect();

assert_eq!(v2, vec![2, 3, 4]);
```

Notice that we also need to *collect* the results, as otherwise the iterator doesn't run.

https://doc.rust-lang.org/book/ch13-02-iterators.html#methods-that-produce-other-iterators

Iterator filter

Another useful function is *filter*, which is like map but uses the bool result to filter the iterator.

```
fn shoes_in_my_size(shoes: Vec<Shoe>, shoe_size: u32) -> Vec<Shoe> {
    shoes.into_iter().filter(|s| s.size == shoe_size).collect()
}
```

This is a fairly complex one-liner, that first does a move into an iterator, then runs a filter on each value and compares shoe size, finally collecting the results into a new vector and returning it.

https://doc.rust-lang.org/book/ch13-02-iterators.html#using-closures-that-capture-their-environment

Custom Iterator

We need two things for a custom iterator:

- *Item* type
- *next* method

```
struct Counter {
    count: u32,
}
impl Counter {
    fn new() -> Counter {
        Counter { count: 0 }
    }
}

impl Iterator for Counter {
    type Item = u32; // required for Iterator
    fn next(&mut self) -> Option<Self::Item> {
        if self.count < 5 {
            self.count += 1;
            Some(self.count)
        } else {
            None
        }
    }
}
```

https://doc.rust-lang.org/b
with-the-iterator-trait

Placeholder Types

The *Item* type in the *Iterator* trait is called a placeholder type, and is conceptually very similar to generics. Indeed, you can accomplish iteration with generics.

But the reason to use a placeholder type is to force a single type on the iterator, so there is only one iterator for a type. This also means we need fewer type annotations.

https://doc.rust-lang.org/book/ch19-03-advanced-traits.html#specifying-placeholder-types-in-trait-definitions-with-associated-types

Operator Overloading

While you can't create or overload arbitrary operators, the ones with traits defined in std::ops are possible.

If we want to overload the + operator, we would implement the *Add* trait.

```
impl Add for Point {
    type Output = Point;
    fn add(self, other: Point) -> Point {
        Point {
            x: self.x + other.x,
            y: self.y + other.y,
        }
    }
}
```

https://doc.rust-lang.org/book/ch19-03-advanced-traits.html#default-generic-type-parameters-and-operator-overloading

Operator Overloading

This is great for operators between the same type. And that's the default, but not the only option.

Take a look at how the *Add* trait is defined:

```
trait Add<Rhs=Self> {
    type Output;

    fn add(self, rhs: Rhs) -> Self::Output;
}
```

Notice the generic type *Rhs*, which defaults to *Self.*

https://doc.rust-lang.org/book/ch19-03-advanced-traits.html#default-generic-type-parameters-and-operator-overloading

Operator Overloading

We can use this to specify other types as well:

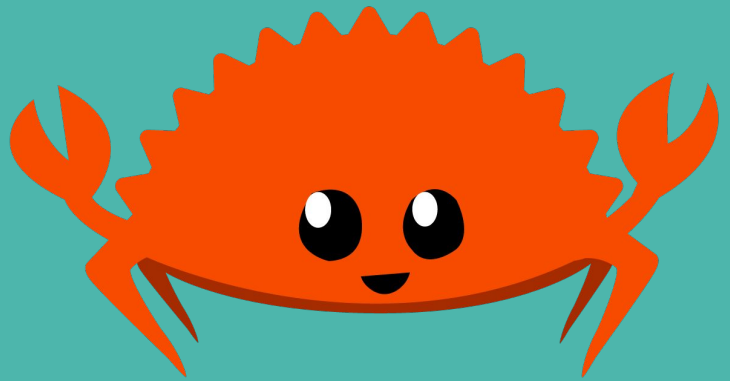
```
struct Millimeters(u32);
struct Meters(u32);

impl Add<Meters> for Millimeters {
    type Output = Millimeters;

    fn add(self, other: Meters) -> Millimeters {
        Millimeters(self.0 + (other.0 * 1000))
    }
}
```

https://doc.rust-lang.org/book/ch19-03-advanced-traits.html#default-generic-type-parameters-and-operator-overloading

End of Lecture



Exercises

1. Fix the lifetime errors in exercises/day7/check_errors.

2. Construct a convertible unit hierarchy for measuring distance. Stub code is provided in exercises/day7/distance.