Lifetimes

This lesson introduces you to an important concept, i.e., lifetime.

We'll cover the following

What Is a Lifetime?

Lifetimes describes the scope that a reference is valid for.

Let's look at a real-life analogy to explain this concept. You are the owner of a book, but you allow other people to use it for some time.

Let's understand this concept in terms of memory management. Having a reference to a resource that someone else has the ownership of can be complicated. For example, imagine this set of operations:

- I have some kind of resource.
- I allow you to borrow the resource.
- I decide my task is completed with the resource, and I want to deallocate it, while you still have the reference of my resource.
- You still want to use the resource.

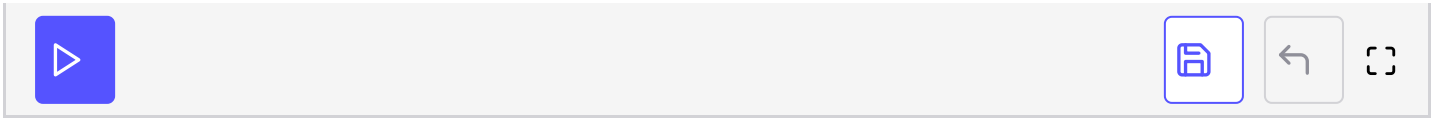
Now you are referencing a resource that is deallocated, an invalid resource. This is called a **dangling pointer** or **use after free**, when the resource is memory.

To make sure that step four never happens, the concept of a lifetime comes into play!

There are two cases to consider.

Case 1: When we know the lifetime of a variable by just looking at the program code

```
#![allow(dead_code)]
struct Course{
   name: String,
   id : i32,
}
fn main(){
  let c1:&Course;
    {
    let c2: Course = Course {
      name : String::from("Rust"),
      id:101,
    };
    }
    c1 = &c2; // allocated reference to a memory location that is dropped
}
```



The above example gives an error, ✕, since `c1` references `c2` which is deallocated when the scope of `c2` finishes.

It is implicit that the scope of `c2` finishes and it cannot be used after **line 13**. But when there is a function that takes an argument by reference, we can be either implicit or explicit about the lifetime of the reference.

Case 2: When we can't say anything about the lifetime of a variable

```
#![allow(dead_code)]
struct Course{
   name: String,
   id : i32,
}
fn get_course(c1: &Course, c2: &Course)->&Course{
  if c1.name == "Rust" {
     c1
  }
  else {
     c2
  }
}
fn main(){
  let c1: Course = Course {
      name : String::from("Rust"),
      id:101,
```

```
    };

   let c2: Course = Course {
      name : String::from("C++"),
      id:101,
    };
    get_course(c1, c2);
}
```

The above code gives an error, ✕, which says missing lifetime specifier. In the function definition, `c1` is referencing Course and `c2` is referencing Course. What if this particular piece of memory is invalid? And for how long does it remains valid? How can we ensure that the function is going to return a valid reference? The compiler is confused what the lifetime of the variable is. Solution: use the *lifetime annotation*.

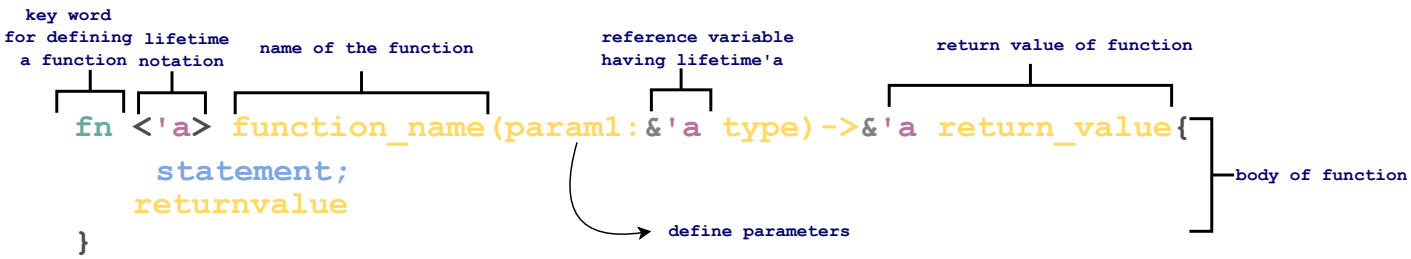
> **How does the compiler know the lifetime of a variable?**
>
> Rust compiler has a **borrow checker**. It compares the scope of the reference variable and the variable being referred. It ensures that the variable being referred to lives as long as the variable referencing it.

Lifetime Annotation

It describes the lifetime of a reference.

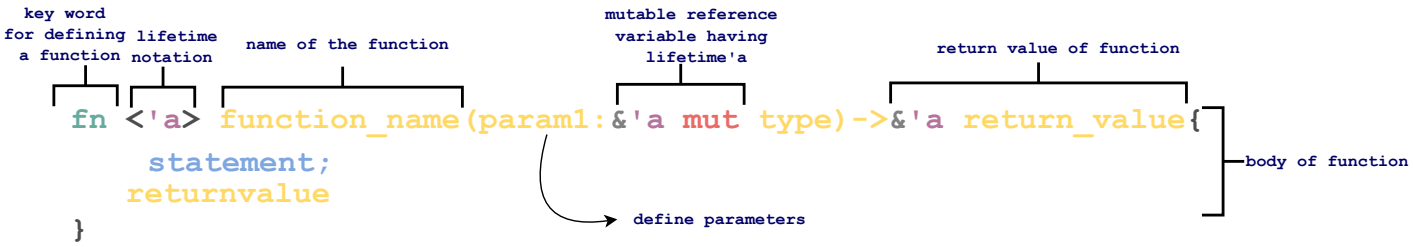
- Starts with an apostrophe
- Followed by a small case name, generally a single letter

Function With Reference Variable Having a Lifetime



Defining a function with reference having a lifetime

Function With Mutable Reference Variable Having a Lifetime



Defining a function with mutable reference having a lifetime

Example

Now, coming back to the example in Case 2. The following code annotates the lifetime of variable `c1`, `c2`, and the return value.

```
#![allow(dead_code)]
struct Course{
   name: String,
   id : i32,
}
fn get_course<'a> (c1: &'a Course, c2: &'a Course) -> &'a Course {
  if c1.name == "Rust" {
     c1
  }
  else {
     c2
  }
}
fn main(){
  let c1: Course = Course {
      name : String::from("Rust"),
      id:101,
    };

   let c2: Course = Course {
      name : String::from("C++"),
      id:101,
    };
    get_course(&c1, &c2);
}
```

Here, lifetime annotation is used for parameter `c1` and `c2` meaning both parameters have the same lifetime as that of the function. Basically, it is trying to tell the compiler that there is a lifetime for which both of the references are valid and for the same lifetime the reference for the return type is also valid. Earlier the issue was that the time for which the return type was valid was unknown. Now, the borrow checker can check whether the reference is valid or not and gives us a

compile-time error if it is invalid. Here, in this case, the reference is valid for `'a`.

> The lifetime annotation `<'a>` is basically a relationship. It is not used standalone. If you don't write this, the code will be correct syntactically, but the compiler will generate a compile-time error, ✕.

> **How is `'a` calculated?**
>
> `'a` refers to the lifetime in which all the references are valid, which is the overlap of the related lifetimes, or the common lifetime of n number of instances. As a result, lifetime having the smallest reference of all references get assigned to `'a.`

Multiple Lifetimes

There are two possibilities when the reference variables are used as function parameters:

Multiple references have the same lifetime

```
fn fun_name <'a>(x: & 'a i32 , y: & 'a i32) -> & 'a i32
  // statements
```

Here, the variable `x` and `y` have the same lifetime `'a`.

Multiple references have different lifetimes

```
fn fun_name<'a , 'b>(x: & 'a i32 , y: & 'b i32)
 // statements
```

Here, both references `x` and `y` have different lifetimes. `x` has lifetime `'a` and `y` has lifetime `'b`.

Now that you have learned about lifetimes, let's learn about lifetime elision in the next lesson.