

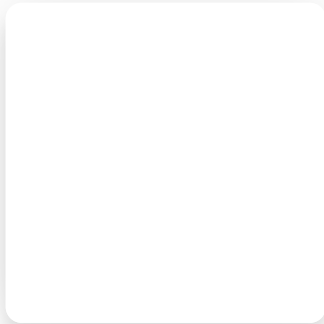
Can't get enough? Subscribe to The Replay, our weekly newsletter



Understanding lifetimes in Rust

August 27, 2020 · 6 min read

So you've decided to take another crack at this Rust thing. Last time went pretty smoothly, except for some minor hiccups with the [borrow checker](#). But you got through it and gained a better understanding of how it works in the process. Maybe it'll all be worth it in the end?



Rust with Luca Palmieri

PREVIEW Jul 2021 · PodRocket - A web

Follow

You've got some grand plans and you're not going to let the borrow checker stop you. You can practically feel the energy coursing through your veins as you imprint your thoughts on the keyboard and translate them into pure Rust. This must be that sweet feeling you've heard so much about.

You save your project, start the compilation process, and...

```
error[E0597]: `x` does not live long enough
```

You sigh. Not this again.

You take a deep breath, lower your shoulders, and read the error message one more time. "Does not live long enough." What does that even mean?

You've encountered another one of Rust's peculiarities: lifetimes.

Introducing lifetimes

Lifetimes are what the Rust compiler uses to keep track of how long references are valid for. Checking references is one of the borrow checker's main responsibilities. Lifetimes help the borrow checker ensure that you never have invalid references.

Lifetime annotations enable you to tell the borrow checker how long references are valid for. In many cases, the borrow checker can infer the correct lifetimes and take care of everything on its own. But often it needs your help to figure it out.

In this guide, we'll go over the basics of lifetimes and annotations and demonstrate how to work with them. We'll also look at some common scenarios you might run into and walk through how to solve them with lifetimes. To follow along, you should have a basic grasp of Rust and some of its concepts (such as the borrow checker), but nothing particularly deep.

Lifetime annotations

Before we go any further, just a short note on the notation of lifetimes since it's a bit different from what you get in a lot of other languages.

Lifetimes are annotated by a leading apostrophe followed by a variable name. When talking about generic lifetimes, we often use single, lowercase letters, starting from `'a` , `'b` , etc. However, there is nothing stopping you from using longer, more explanatory names if that suits you better.

Why we need lifetimes

If it's such a weird feature, then why do we need lifetimes? The answer lies in Rust's ownership model. The borrow checker takes care of allocating and freeing memory and also ensures that no references point to memory that has been freed. Like borrows, lifetimes are checked at compile time, which means your program can't compile if the borrow checker deems the references invalid.

Over 200k developers use LogRocket to create better digital experiences

Learn more →

In particular, lifetimes are important to keep in mind when returning references from functions and when creating structs with references. These are both common situations, and it's easy to get lost if you don't understand what's going on.

The explanatory example

Ultimately, lifetimes are a matter of scope. Values get dropped when they go out of scope and any references to them after they have been dropped are invalid.

The simplest way to demonstrate lifetimes is something like the following example, shamelessly stolen/adapted from [the official book's chapter on lifetimes](#).

```
// this code sample does *not* compile
{
    let x;
    {
        // create new scope
        let y = 42;
        x = &y;
    }
    // y is dropped

    println!("The value of 'x' is {}. ", x);
}
```

This little piece of code has two distinct scopes. When the inner scope closes, `y` is dropped. At that point, even if `x` is still available in the outer scope, the reference is invalid because the value it pointed to is dropped; the value that `x` points to “does not live long enough.”

In lifetime jargon, we can say that the outer scope has the lifetime `'outer` and the inner scope the lifetime `'inner`. `'outer` clearly outlives `'inner` in this case. When `'inner` ends, all values with that lifetime are invalidated.

Lifetime elision

When writing functions that accept references as arguments, the compiler can infer the correct lifetimes in many cases, saving you the trouble of writing them out by hand. When lifetime annotations are implicit, we call this lifetime elision.

The compiler uses three rules to figure out whether lifetime annotations can be elided or not. The [book's section on lifetime elision](#) talks about these rules in detail, but the short form is that you can elide lifetime annotations in functions if one of the following is true.

- The function doesn't return a reference
- There is exactly one reference input parameter
- The function is a method, taking `&self` or `&mut self` as the first parameter

Examples and common problems

Lifetimes are tricky to wrap your head around, and it's unlikely that a wall of text will really help you understand how they work. The best way to get a proper understanding is, of course, to play around with lifetimes yourself and solve problems. That said, a couple of examples can go a long way.

Returning references from functions

You can't return a reference from a function without also passing in a reference. If you try, you'll find that the reference is invalid as soon as the function returns and your program won't compile. You could use the `'static` lifetime, but that's probably not what you want. It's also outside the scope of this article, so let's forget about it for now.

More great articles from LogRocket:

- Don't miss a moment with [The Replay](#), a curated newsletter from LogRocket
 - [Learn](#) how LogRocket's Galileo cuts through the noise to proactively resolve issues in your app
 - Use React's `useEffect` to [optimize your application's performance](#)
 - Switch between [multiple versions of Node](#)
 - [Discover how to animate](#) your React app with AnimXYZ
 - [Explore Tauri](#), a new framework for building binaries
 - Compare [NestJS vs. Express.js](#)
-

If your function takes exactly one reference parameter, then you'll be fine without annotations. All output references will be given the same lifetime as the input parameter. As such, this simple function will compile just fine, even if there are no explicit lifetime annotations.

```
fn f(s: &str) -> &str {  
    s  
}
```

However, if you add another input string parameter (even if you don't use it), you suddenly won't be able to compile this:

```
// this code sample does *not* compile  
fn f(s: &str, t: &str) -> &str {  
    if s.len() > 5 { s } else { t }  
}
```

That's because of how the automatic lifetime annotation works. When a function accepts multiple references, they're each given their own lifetime. We know that the returned reference must be one of the references we received as an input argument, but we don't know which one. What goes in place of the '???' below?

```
// this code sample does *not* compile  
fn f<'a, 'b>(s: &'a str, t: &'b str) -> &'??? str {  
    if s.len() > 5 { s } else { t }  
}
```

Imagine that you want to use the returned value outside of this function. What lifetime would you assign to it? **The only guarantee is that the reference you return is valid for at least as long as the shortest-lived reference you pass into the function.** That tells the compiler that these two references are definitely valid for the shorter lifetime. Nothing is guaranteed outside of that.

The way to achieve this is to give both input parameters the same lifetime annotation. It's how we tell the compiler that as long as both of these input parameters are valid, so is the returned value.

```
fn f<'a>(s: &'a str, t: &'a str) -> &'a str {  
    if s.len() > 5 { s } else { t }  
}
```

If you're returning a reference from a function that takes multiple input lifetime parameters but you know exactly which one you're returning, you can annotate that specific lifetime. That way, the relationship between the lifetimes doesn't matter.

```
fn f<'a, 'b>(s: &'a str, _t: &'b str) -> &'a str {  
    s  
}
```

Structs with references

References in structs can be a real hassle. You're often better off avoiding them and using owned values instead. That way, you don't need to worry about references being invalidated and lifetimes not lasting long enough. In my experience, it's usually also what you want.

However, there are certain cases where structs with references are exactly what you want — in particular, if you want to create a view into something else. Using structs with references is a great way to organize some data into a package that's easier to handle without moving or copying data. This means that the original data source can still be referenced elsewhere and you're spared the hassle of cloning the data.

For example, let's say you want to find the first and the last sentence of a paragraph and keep them in a struct `S`. Because you don't want to copy the data, you need to use references and give them lifetime annotations.

```
struct S<'a> {  
    first: &'a str,  
    last: &'a str,  
}
```

You could use a function like this to populate the struct. For simplicity's sake, we'll assume that a full stop is the only sentence-ending punctuation mark in use. If the paragraph is empty, return `None`, and if there is only a single sentence, use that as both the first and the last sentence.

```
fn try_create(paragraph: &str) -> Option<S> {  
    let mut sentences = paragraph.split('.').filter(|s|  
    !s.is_empty());  
    match (sentences.next(), sentences.next_back()) {  
        (Some(first), Some(last)) => Some(S { first, last }),  
        (Some(first), None) => Some(S { first, last: first }),  
        _ => None,  
    }  
}
```

You don't need to annotate lifetimes in the function signature because the compiler can figure it out for you. In a case like this, there is really only one choice: the lifetime of the input string. Pretty neat, huh?

Summary and further reading

This has been a cursory glance at lifetimes and lifetime annotations. We glossed over a lot of the finer and more intricate details of how lifetimes work, but we covered enough ground that you should be able to reason about them when you run into an issue.

Because lifetimes are such an important part of Rust, I encourage you to read the “[Validating References with Lifetimes](#)” chapter of “[The Rust Programming Language](#)” for a more comprehensive introduction.

Furthermore, if you feel like you've got a decent grasp on lifetimes but want to dive a bit deeper, check out Jon Gjengset's excellent video, “[Crust of Rust: Lifetime Annotations](#),” where he explores a case that needs multiple explicit lifetime annotations. He also gives a great introduction to lifetime annotations in general, so it's well worth a watch just for that.