한 번에 끝내는 블록체인 개발 A to Z

Chapter 2

Rust Advanced

Chapter 2 Rust Advanced

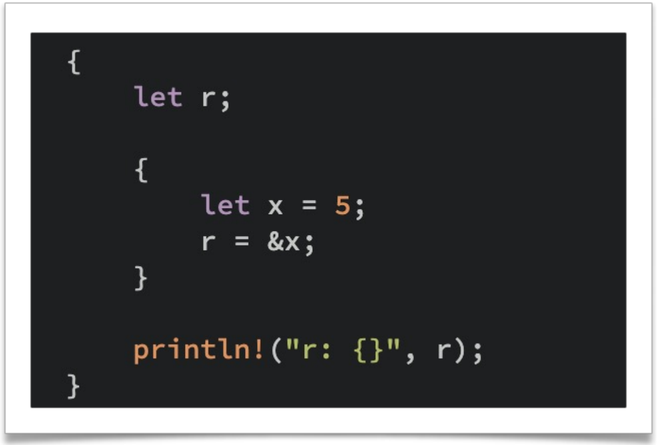
Validating References with Lifetimes

Lifetimes

Lifetimes are another kind of generic that we've already been using. Rather than ensuring that a type has the behavior we want, lifetimes ensure that references are valid as long as we need them to be.

every reference in Rust has a lifetime, which is the scope for which that reference is valid. Most of the time, lifetimes are implicit and inferred, just like most of the time, types are inferred.

Dangling Reference

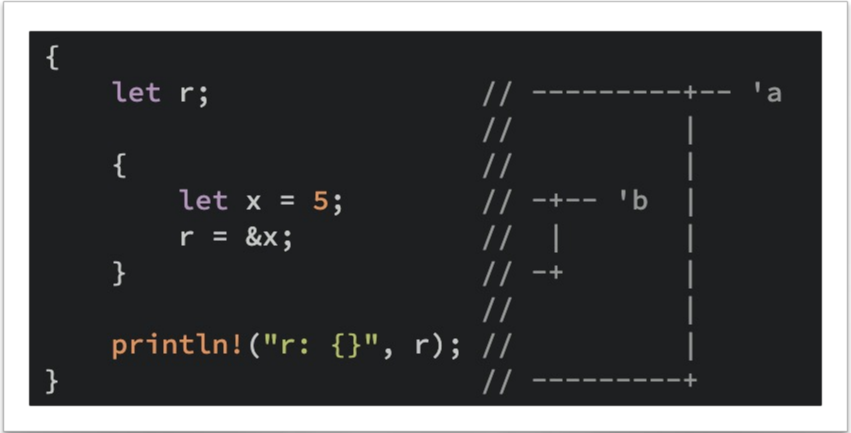


```
{
    let r;

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

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

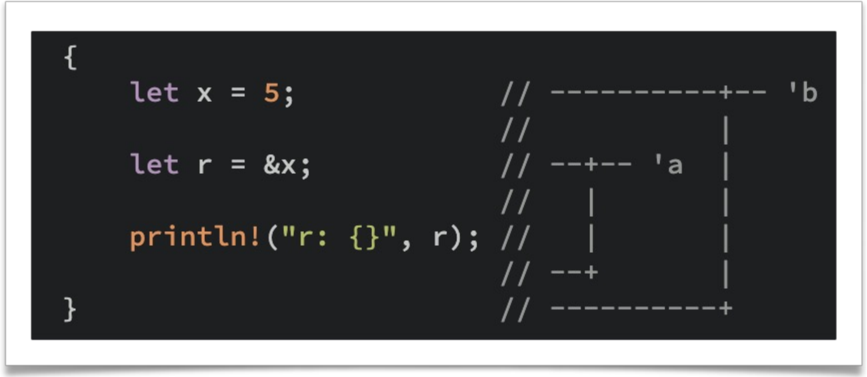
Borrow Checker

The Rust compiler has a borrow checker that compares scopes to determine whether all borrows are valid.

We’ve annotated the lifetime of **r** with **'a** and the lifetime of x with **'b**. As you can see, the inner **'b** block is much smaller than the outer **'a** lifetime block. At compile time, Rust compares the size of the two lifetimes and sees that **r** has a lifetime of **'a** but that it refers to memory with a lifetime of **'b**. The program is rejected because **'b** is shorter than **'a**: the subject of the reference doesn’t live as long as the reference.



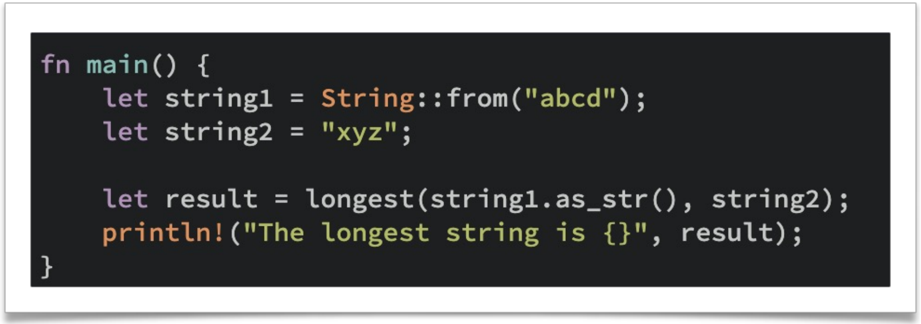
```
{
    let r;                // ---------+-- 'a
                          //          |
    {                     //          |
        let x = 5;        // -+-- 'b  |
        r = &x;           //  |       |
    }                     // -+       |
                          //          |
    println!("r: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



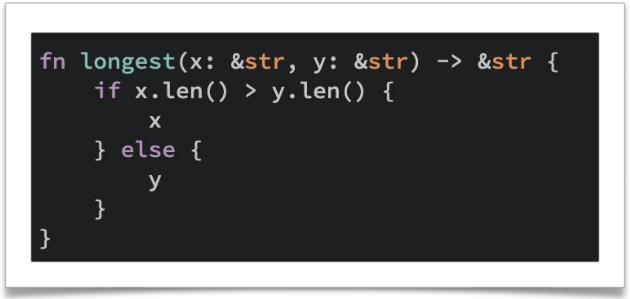
```
{
    let x = 5;            // ----------+-- 'b
                          //           |
    let r = &x;           // --+-- 'a  |
                          //   |       |
    println!("r: {}", r); //   |       |
                          // --+       |
}                         // ----------+
```

Generic Lifetimes in Functions

We'll write a function that returns the longer of two string slices. This function will take two string slices and return a single string slice.



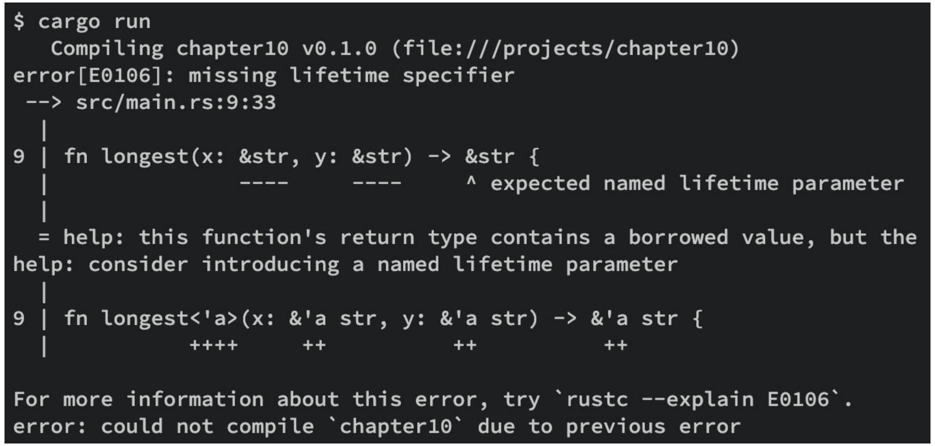
```rust
fn main() {
    let string1 = String::from("abcd");
    let string2 = "xyz";

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



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

Generic Lifetimes in Functions (2)

We can’t look at the scopes to determine whether the reference we return will always be valid. **The borrow checker can’t determine this either, because it doesn’t know how the lifetimes of x and y relate to the lifetime of the return value.** To fix this error, we’ll add generic lifetime parameters that **define the relationship between the references** so the borrow checker can perform its analysis.

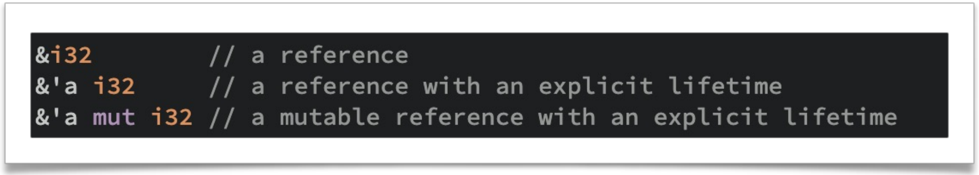


```
$ cargo run
   Compiling chapter10 v0.1.0 (file:///projects/chapter10)
error[E0106]: missing lifetime specifier
 --> src/main.rs:9:33
  |
9 | fn longest(x: &str, y: &str) -> &str {
  |               ----     ----     ^ expected named lifetime parameter
  |
  = help: this function's return type contains a borrowed value, but the
help: consider introducing a named lifetime parameter
  |
9 | fn longest<'a>(x: &'a str, y: &'a str) -> &'a str {
  |           ++++     ++          ++          ++

For more information about this error, try `rustc --explain E0106`.
error: could not compile `chapter10` due to previous error
```

Lifetime Annotation

Lifetime annotations don’t change how long any of the references live. Rather, they describe the relationships of the lifetimes of multiple references to each other without affecting the lifetimes.

Lifetime annotations have a slightly unusual syntax: the names of lifetime parameters must start with an apostrophe (') and are usually all lowercase and very short, like generic types.

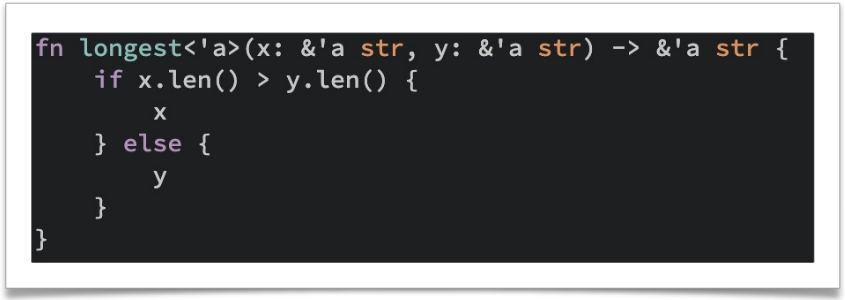


```
&i32        // a reference
&'a i32     // a reference with an explicit lifetime
&'a mut i32 // a mutable reference with an explicit lifetime
```

Lifetime Annotation in Function

As with generic type parameters, we need to declare generic lifetime parameters inside angle brackets between the function name and the parameter list.

We want the signature to express the following constraint: the returned reference will be valid as long as both the parameters are valid.

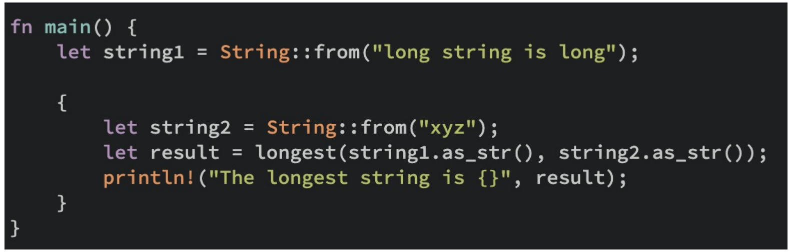


```rust
fn longest<'a>(x: &'a str, y: &'a str) -> &'a str {
    if x.len() > y.len() {
        x
    } else {
        y
    }
}
```

Lifetime Restriction

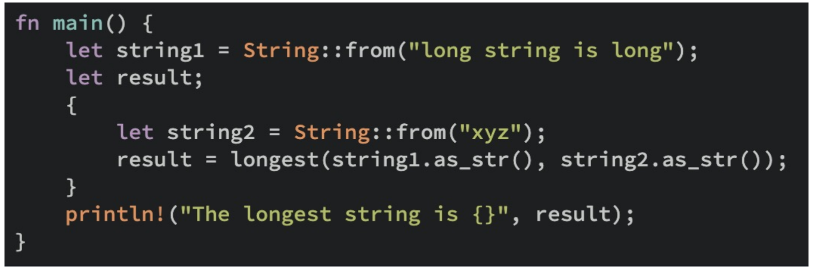
When we pass concrete references to longest, the concrete lifetime that is substituted for **'a** is the part of the scope of x that overlaps with the scope of y. In other words, the generic lifetime **'a** will get the concrete lifetime that is **equal to the smaller of the lifetimes of x and y**. Because we've annotated the returned reference with the same lifetime parameter **'a**, the returned reference will also be valid for the length of **the smaller of the lifetimes of x and y**.

Valid Usage



```rust
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);
    }
}
```

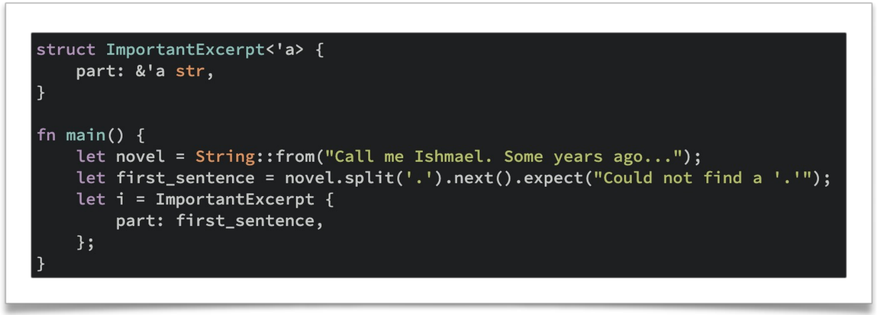
Invalid Usage



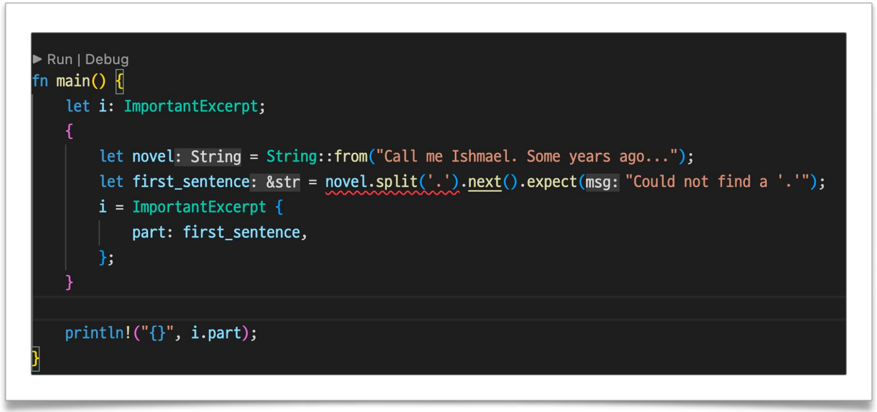
```rust
fn main() {
    let string1 = String::from("long string is long");
    let result;
    {
        let string2 = String::from("xyz");
        result = longest(string1.as_str(), string2.as_str());
    }
    println!("The longest string is {}", result);
}
```

Lifetime Annotation in Struct

This struct has one field, part, that holds a string slice, which is a reference. As with generic data types, we declare the name of the generic lifetime parameter inside angle brackets after the name of the struct so we can use the lifetime parameter in the body of the struct definition. This annotation means an instance of ImportantExcerpt can't outlive the reference it holds in its part field.



```rust
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,
    };
}
```

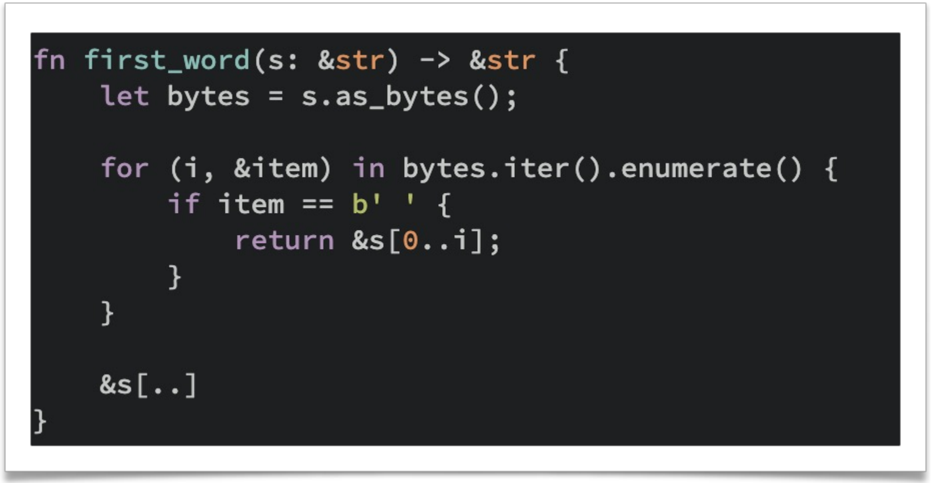


```rust
▶ Run | Debug
fn main() {
    let i: ImportantExcerpt;
    {
        let novel: String = String::from("Call me Ishmael. Some years ago...");
        let first_sentence: &str = novel.split('.').next().expect(msg: "Could not find a '.'");
        i = ImportantExcerpt {
            part: first_sentence,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

    println!("{}", i.part);
}
```

Lifetime Elision

The reason this function compiles without lifetime annotations is historical. After writing a lot of Rust code, the Rust team found that Rust programmers were entering the same lifetime annotations over and over in particular situations. These situations were predictable and followed a few deterministic patterns. The developers programmed these patterns into the compiler's code so the borrow checker could infer the lifetimes in these situations and wouldn't need explicit annotations.

The patterns programmed into Rust's analysis of references are called the lifetime elision rules.

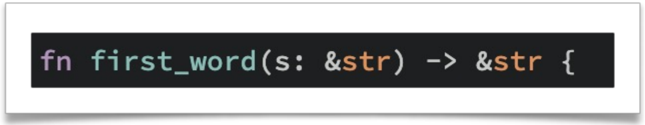


```rust
fn first_word(s: &str) -> &str {
    let bytes = s.as_bytes();

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

    &s[..]
}
```

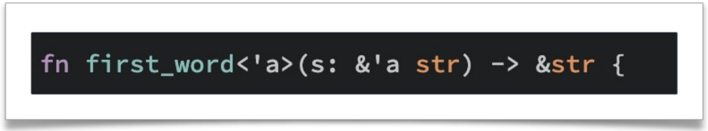
Lifetime Elision (2)

The compiler uses **three rules** to figure out the lifetimes of the references when there aren't explicit annotations.

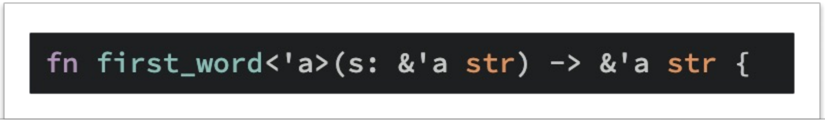
1. The compiler assigns a lifetime parameter to each parameter that's a reference.
2. If there is exactly one input lifetime parameter, that lifetime is assigned to all output lifetime parameters.
3. If there are multiple input lifetime parameters, but one of them is **&self** or **&mut self** because this is a method, the lifetime of self is assigned to all output lifetime parameters.



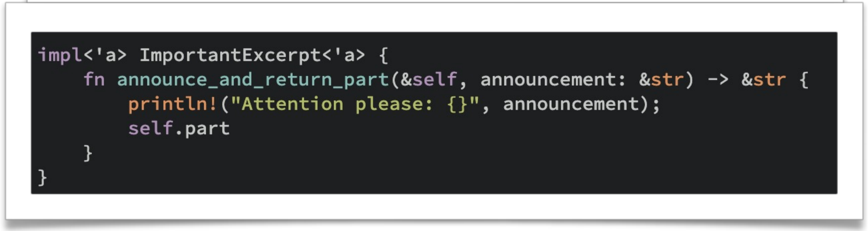
```rust
fn first_word(s: &str) -> &str {
```



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



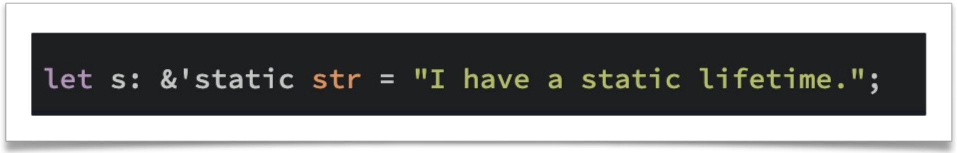
```rust
fn first_word<'a>(s: &'a str) -> &'a str {
```



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

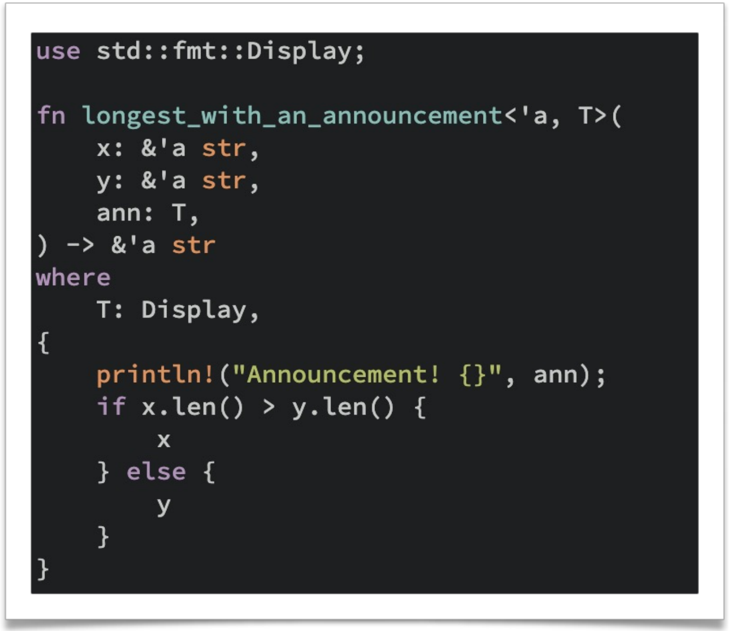
The Static Lifetime

One special lifetime we need to discuss is **'static**, which denotes that the affected reference can live for the entire duration of the program.



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

Trait Bounds, and Lifetimes



```rust
use std::fmt::Display;

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
    }
}
```