Lifetime Elision

This lesson discusses lifetime elision in Rust.

We'll cover the following

What Is Lifetime Elision?

Some common coding patterns were identified and annotating them was an extra effort. In order to avoid that overhead, Rust allows lifetimes to be elided or omitted. This is known as lifetime elision. Rust does so by designing rules for omitting lifetime annotation. They are tested at compile time and are used to determine a lifetime.

Lifetime Ellison can appear in two ways:

- An **input lifetime** is a lifetime associated with a parameter of a function.

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

  Here, the **input lifetime of `x` is `'a`**.

- An **output lifetime** is a lifetime associated with the return value of a function.

  ```
  fn fun_name<'a>() -> & 'a i32;
  ```

  Here, the **output lifetime of return value is `'a`**.

> **Note:** A function can have both input and output lifetimes.

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

Here, the `x` has the input lifetime and the return value has the output lifetime.

Rules for Elision

Elision rules are as follows:

Rule # 1

Each input parameter gets its own lifetime. If the lifetime is not specified, then the lifetime of each parameter is different.

Elided lifetime:

```
fn fun_name( x: &i32, y: &i32){
}
```

Expanded form:

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

Rule # 2

If there is only one input parameter, its lifetime is assigned to all the elided output lifetimes.

Elided lifetime:

```
fn fun_name(x: i32) -> &i32{
}
```

Expanded form:

```
fn fun<'a>(x: i32) -> & 'a i32 {
}
```

Rule # 3

If there are multiple input lifetimes, one of them is `&self` or `&mut self`, the lifetime of `self` is assigned to all elided output lifetimes.

Elided lifetime:

```
fn fun_name(&self, x : &str) -> & str
 {
}
```

Expanded form:

```
fn fun_name<'a,'b>(& 'a self,
 x : & 'b str) -> & 'a str {
}
```

> **Note:** If the compiler can't infer the lifetime, it throws an error.

Now that you have learned everything about memory management in Rust using ownership, borrowing and lifetime, let's conclude the discussion on Rust in the next chapter.