
RUST PROGRAMMING
Lifetime and References
Kamal kumar mukiri, Apt Computing Labs

CONTENTS

Dangling pointer


```
1  fn main() {
2      let r: &i32;
3      {
4          let x: i32 = 42;
5          r = &x;
6      }
7      println!("r: {}", r);
8  }
```

```
  --> Simple.rs:5:13
   |
4  |          let x = 42;
   |              - binding `x` declared here
5  |          r = &x;
   |              ^^ borrowed value does not live long enough
6  |      }
   |      - `x` dropped here while still borrowed
7  |      println!("r: {}", r);
   |                        - borrow later used here
```

What are “lifetimes” ?

- A **lifetime** is a way to describe how long a reference is valid.
- Every reference in Rust has a **lifetime**, either explicitly declared or inferred by the compiler.
- Lifetimes are most commonly used when working with references in structs, functions, or methods to ensure memory safety.

'static lifetime:


```
fn static_lifetime_example() -> &'static str {
    "Hello, static lifetime!" // This is valid
}

fn main() {
    let msg = static_lifetime_example();
    println!("{}", msg);
}
```

- `'static` is a special, predefined lifetime.
- It means the reference is valid for the entire duration of the program.
- Common examples:
  - String literals: `"Hello, World!"` has a `'static` lifetime because it is stored in the program's binary and exists for the program's lifetime.

’static lifetime:

You can use **'static** for references in structs when the data lives for the entire program.

```
struct Config {
    data: &'static str, // Data must live for the program's lifetime
}

fn main() {
    let config = Config {
        data: "App settings", // String literal with 'static lifetime
    };
    println!("{}", config.data);
}
```

Named lifetime (Eg: `'a`):

Lifetime Annotations in Functions

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

fn main() {
    let str1 = String::from("Hello");
    let str2 = "World!";
    let result = longest(&str1, &str2); // The retu
    println!("The longest string is: {}", result);
}
```

- Named lifetimes are user-defined and allow more flexibility compared to 'static. They express relationships between the lifetimes of references.
- **Syntax**
  - <'a>: Declares a lifetime parameter named 'a.
  - &'a T: A reference with lifetime 'a.

Named lifetime (Eg: `'a`):

Lifetimes in Structs

```
struct Book<'a> {
    title: &'a str, // Reference must outlive 'a
}

fn main() {
    let title = String::from("Rust Programming");
    let book = Book {
        title: &title, // Reference tied to `title`
    };
    println!("Book title: {}", book.title);
}
```

- **Note:** If a struct type contains references, you must name those references' lifetimes.
- Here, the lifetime 'a ensures the reference title in Book does not outlive the String it refers to.

Notes on explicit Lifetime:


```
fn invalid_reference<'a>() -> &'a i32 {
    let x = 42;
    &x // Error: `x` does not live long enough
}
```

- **Note:** Rust prevents returning a reference to a local variable because it would be invalid once the function exits.

Notes on explicit Lifetime:

- Lifetimes are a core feature of Rust’s memory safety.
- `'static`: For data that lives for the program's lifetime.
- `'a, 'b`, etc.: User-defined lifetimes to define relationships between references.
- Lifetimes ensure references are valid, preventing dangling or invalid references.