Lifetimes

Learn to Code with Rust / Section Review

Lifetimes

- A **lifetime** is the region of code that a value is alive, which means functional, valid, useful.
- A **lifetime** is the time during which a value exists at a particular memory address.
- Lifetimes exist so that the borrow checker can identify **dangling references**.
- A **dangling reference** is a reference to data that does not exist.

Scopes and Lifetimes

- A **scope** is a region of code belonging to a block (a section between a pair of opening and closing curly braces).
- A value's lifetime is often connected to its scope.
- When an owner reaches the end of its scope, it cleans up the associated data. The value's lifetime ends.

References and Lifetimes

- The **referent** is the data/value that a **reference** borrows.
- A reference's lifetime ends at the last place in the code where it is used (**non-lexical scope**).
- A reference's lifetime is connected to the lifetime of its **referent**, the source of its data.
- The reference's lifetime must be contained within the referent's lifetime to avoid a dangling reference.

Generic Lifetimes vs. Concrete Lifetimes

- A **concrete lifetime** is the region of code that a value exists in the program, which corresponds to the time that it lives in its memory address.
- A **generic lifetime** is a hypothetical lifetime, a non-specific lifetime, a future lifetime that can vary.

Lifetime Annotations

- Lifetime annotations declare generic lifetimes.
- Lifetime annotations identify relationships between reference parameters and reference return values.
- An annotation creates a coupling between the return reference's lifetime and the referent's lifetime.

Lifetime Syntax

- Use a pair of angle brackets.
- Assign the lifetime a short name starting with a tick. **'a** is a common choice.
- Use the lifetime to mark the connections between references and the return value.
- Think of the lifetime as augmenting a type. A type of **&str** is different from a type of **&'a str**.

Lifetime Syntax II

- Marking multiple reference parameters with the same lifetime annotation does not mean they *have* to have an identical lifetime.
- If there are different concrete lifetimes, the returned reference must live within the *overlapping* lifetime (the shorter of the two lifetimes).
- The borrow checker will validate that a returned reference to *either* parameter will be valid.

Lifetime Elision Rules

1. The borrow checker assigns a separate lifetime to each reference parameter.

2. If there is one reference parameter and the return value is a reference, the borrow checker infers that the parameter's lifetime will apply to the return value.

3. In a method definition, if the parameter is a reference to the instance (**&self**), the borrow checker assigns the lifetime of self (the instance) to the return value.

Lifetimes in Structs

- A struct can store a reference in a field.
- The lifetime of the struct must be connected with the lifetime of the field.
- The lifetime of the struct must end before the lifetime of the field's referent.
- Otherwise, the struct's field would be a dangling reference.