Lifetimes

Borrow Checker

- Remember dangling references from the Ownership chapter?
- Remember that dangling references occur when you reference invalid data
- The example we used then was to return a reference that only existed in the scope of a function
- Once out of scope that reference is dangling
- This can apply to any scope like **expression blocks**
- Here is a simple example:

```
let mut r;

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

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

  - You will see an error that x Doesnt "live long enough" meaning that the value of x is dropped before r can be printed
  - This is a simple example of a dangling reference situation

- Rust checks for dangling references and borrow issues at compile time using the **borrow checker**
- It specifically checks for the **lifetime** of variables when they are borrowed
- To make this more clear check this:

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

  - Now we can see the lifetime of our variables

- Here `r` is following the `'a` lifetime and `x` is following the `'b'` lifetime
- However since `r` points to `x` at the end of the expression block, the lifetime of `r` becomes the same as the lifetime of `x`.
- That way we can clearly see that `r` is invalid to use at the println function

- Let's take a look at an example where it can work out. In other words, use `r` as a reference to `x` while `x` is still *alive*

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

- Here we can see the lifetime of the variables in a case that does work out
- Since `x` exists in the entire lifetime it's reference is being used, there is no issue
- We also see that the lifetime of `r` is the same as the lifetime of `x`

- The borrow checker can perform all these lifetime checks without our help

Generic Lifetime Annotations

- The borrow checker is pretty good at its job, but the rules of lifetimes are confusing and we can help make them more clear for the borrow chekcer

- And for that we need generic lifetime annotations

- Let's throw a function into this and some parameters

- Let's say we want a function to return the longest string between two strings

```
fn longest(string1: &str, string2: &str) -> &str{
    if string1.len() > string2.len(){
        return string1;
    }
    else{
        return string2
    }
}

let string1 = String::from("Hello, world!");
let string2 = String::from("Hello!");

let result = longest(string1.as_str(), string2.as_str());
println!("result = ", result);
```

- This seems easy enough but actually this throws us an error
    - For one, we do not know the lifetime of the strings. We do a comparison but the lifetime of `string1` and `string2` could be different
        - So returning `string1` might be an error because `string1` lifetime dies out of the function scope and the same issue could happen with `string2`
        - So it's unclear which reference we will return and the lifetime of that reference
    - The borrow checker doesn't know how to handle this ambiguity of the lifetimes

- To help specify the lifetime of the function variables and the return type lifetime we need to use the Generic Lifetime Annotation

- A generic lifetime annotation specifies the lifetime to make it clearer to the borrow checker

- It is denoted by `'a` a tick and generic. it could be `'apple`,`'dog`, `'b`, or anything really

- Like a normal generic, we first define the lifetime with the `<>` after the method name and put our annotation in it

- Once the lifetime is defined, we can provide it to the variables and return value

- Let's fix our function at top

```rust
fn longest<'a>(string1: &'a str, string2: &'a str) -> &'a str{
    if string1.len() > string2.len(){
        return string1;
    }
    else{
        return string2
    }
}

let string1 = String::from("Hello, world!");
let string2 = String::from("Hello!");

let mut result = longest(string1.as_str(), string2.as_str());
println!("result = ", result);
```

    - All the errors should be gone now

    - We can also see that the way to specify the lifetime for a parameter and return type is within the `type` itself.

    - In other words its written as `variable: &'lifetime type`

    - Here are the basic examples:

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

  - We can see that we annotate `string1`, `string2`, and the return value

- **Note** we are not changing the lifetime of anything here

- The annotation **describes the relationship** between all aspects within a scope

- The way to properly read this relationship for the function above is the following:

  - The lifetime of the return reference will be the same as the **smallest** lifeitme of the arguments passed to the function
  - In other words, if `string1` and has a smaller lifetime than `string2`, the return reference will have the same lifetime as `string1`.
  - Regardless of whether `string1` or `string2` is returned, the return reference will only follow the smaller lifetime
  - To make this point more clear, look at the following alteration to our example: `rust let string1 = String::from("Hello, world!2"); { let string2 = String::from("Hello, Everyone!"); result = longest(string2.as_str(), string1.as_str()); println!("result in a new lifetime: {}", result); } println!("result back outside lifetime: {}", result);` * So now we know that `string2` has a different lifetime than `string1` * We can also see that `string2` is the longer string but a shorter lifetime as defined within the inner scope *We will get an error thrown here because although result originally had the lifetime of the main scope, it now has the lifetime of the innerscope with `string2` * So now when `string2` is dropped then so is `result` *The inner scope print will be fine and print `string2` but the outer scope will throw an error * If we were to alter `string2` to make it shorter than `string1` *like so: `let string1 = String::from("Hello to Everyone!");` * Then we would see `string1` printed in the inner scope, but still an error for the outer scope. *That's cause no matter the lifetime of `string1` the `result` takes the shortest lifetime * This means `result` has the same lifetime as `string2` no matter the length of `string2`

- Why does it take the shortest lifetime for the return type?

  - This is because the shortest lifetime would be the *worst case scenario*
  - It is the worst case scenario since you could imagine that if the shortest lifetime was also the return reference, then that gets dropped earlier than the longer lifetimes.
  - If it is dropped earlier but we still want to use it then we will run into a dangling reference
  - So to make it simpler we just want to keep the shortest lifetime possible. Even though if given a longer lifetime then it might work during runtime, the rust compiler needs to be sure at compile time

Thinking in terms of Lifetime

- Now looking back the main issue that we developed here is that we want to pass back a reference after returning from a scope

- This return reference needs to be tied to a lifetime but it only needs to be tied to a lifetime that is a **possible** return reference.

  - In other words, if we take in 3 parameters but only 2 of them are potentially return reference, then only those 2 need to be tied to a lifetime

- Let's see a simple example:

  ```
  fn longest<'a>(string1: &'a str, string2: &str) -> &'a str{
      string1
  }

  let string1 = String::from("Hello, world! 2");
  {
      let string2 = String::from("Hello, Everyone!");
      result = longest(string1.as_str(), string2.as_str());
      println!("result in a new lifetime: {}", result);
  }
  println!("result back outside lifetime: {}", result);
  ```

  - This has no errors because since we do not return `string2` we do not need to tie a lifetime to it
  - And so only `string1` lifeitme is used here which we know can be referenced in both the inner and outer scope

- But we know here that if we remove the lifetime marker from `string1` in the function then we have an error. This is because we return a refernce to it

- So here we meet one crucuial rule of lifetimes, we need to make sure that the return reference is **tied** to at least one argument passed to the function.

- This is because any references created inside the function will be dropped out of the function scope

- References passed to the function however, has a minimum lifetime that's guranteed to be more than the function's scope.

- So tying to the parameter is a necessity when returning a reference from the function

- A quick example:

  ```
  fn longest3<'a>(string1: &str, string2: &str) -> &'a str{
      let s1 = String::from("hi");
      s1.as_str()
  }
  ```

  - First note that this is an error
  - We see that we don't return any of the arugment's references
  - We return a string slice
  - It seems like it should work, but as we saw in the Chapter with Ownership, that this is a dangling reference
  - But thinking about this in terms of lifetime we know that the lifetime of variables declared in the function scope are tied to the function scope
  - So returning a reference from the function scope is a dangling reference

- So to summarize, we need to tie return referneces to arguments in a function and not variables declared in the function scope
- Also note that the talk of lifetime only applies when **borrowing** or scopes, but not when **transferring ownership**
- We performed all these examples with the `&str` type because it performs borrows by default
- However, if we used `String` we would just be transferring ownership and the examples would have been fine
    - Although `&String` would have the same issues as `&str`

Lifetime Annotations in Struct Definitions

- Defining a Struct's lifetime is similar to defining a Generic in a struct just as we saw in the function example

```
struct ImportantExcerpt<'a> {
    part: &'a str,
}

let novel = String::from("Call me Ishmael, Some Years ago...");
let first_sentence = novel.split('.').next().expect("Couldn't find the sentene");
let i = ImportantExcerpt{part: first_sentence};
```

- Previously we used only Owned data types or simple data types in structs
- Now we can declare referenced or borrowed data types within the struct but we need to use the lifetime marker
- Just note that the lifetime of the struct is the same as the shortest lifetime field
- So in our example above, the second `first_sentence` has gone out of scope, we lose access to `i` as a whole

Lifetime Elision

- Ok, so there has been a little misleading information here
- There are some specific cases when returning a reference that the lifetime doesn't need to be explicitly defined
- We had a function called first_word many chapters ago

```
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[..]
}
```

    - Here we can see that we return a string slice but there is no lifetime annotation
    - Sometimes the compiler can deterministically find the lifetime using the **Elision** rules
- Lifetime of arguments passed in to a function are called **input lifetimes** and lifetimes that are returned from the function are called **output lifetimes**
- **Elision Rules**
    - 1st rule: each parameter that is a reference that is passed in gets its own lifetime parameter
    - 2nd Rule: IF there is exactly one input lifetime parameter, then that lifetime parameter is automatically assigned to the output lifetime
    - 3rd Rule: If there are multiple input lifetime parameters, but one of them is `&self`, or `&mut self`, the lifetime of `self` is assigned to all output parameters.
        - This only applies to methods of structs
- The compiler will apply these rules automatically, but if after these three rules are applied and the compiler still doesn't know the lifetime, we need to specify it
- The third rule is important to note that all of its output lifetimes will have the same lifetime as the object that calls it

Lifetime Annotations in Method Definitions

- We discussed the third rule of Elision, so let's see how it looks

```
struct ImportantExcerpt<'a> {
    part: &'a str,
}

impl<'a> ImportantExcerpt<'a> {
    fn level(&self) -> i32 {
        3
    }
}

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

    - Here we can see that we have a Struct with a lifetime described
    - So the first method `level()` we have, we must declare the lifetime annotation for the implementation even if we don't use it
        - At the method level we don't use any annotations for lifetime

    - The second method `announce_and_return_part` returns a `&str` so we would think we need a lifetime annotation
    - However using the third rule of Elision, there is an implicit output lifetime which follows the `&self` lifetime
    - Now the third rule only applies because we don't return any other parameter.
    - If we returned `announcement` argument then we would have an error

Static Lifetimes

- Static Lifetime is a defined lifetime annotation within rust
- This means that your lifetime lasts for the entire duration of the program
    - Remember that lifetime is just meaning how long the memory lasts but it doesn't mean it is globally accessible. The memory may not drop but access into scopes, or privates still take into account
- Many times you will get an error that suggests to make a static lifetime, however this is not recommended as a first action
- It is more recommended to analyze and assign the proper lifetime needed for any particular data
- String literals or string slices are `'static` by default since they are hardcoded to the binary of the file
- The other way of doing a static lifetime is by making a `static` type
- The `static` type is a subset of the `const` type, which means it cannot be changed throughout the program