2_References_en

May 2, 2023

1 References and borrowing

Here is how you would define and use a calculate_length function that has a reference to an object as a parameter instead of taking ownership of the value. The ampersands are **references**, and they allow you to refer to some value without taking ownership of it:

[ ]:
```
fn main() {
    let s1 = String::from("hello");

    let len = calculate_length(&s1);

    println!("The length of '{}' is {}.", s1, len);
}

fn calculate_length(s: &String) -> usize {
    s.len()
}
main();
```

> **Note:** The opposite of referencing by using `&` is **dereferencing**, which is accomplished with the dereference operator, `*`.

We call having references as function parameters **borrowing**. As in real life, if a person owns something, you can borrow it from them. When you're done, you have to give it back.

Just as variables are immutable by default, so are references. We're not allowed to modify something we have a reference to.

[ ]:
```
fn main() {
    let s = String::from("hello");

    change(&s);
}

fn change(some_string: &String) {
    some_string.push_str(", world");
}
main();
```

1.1 Mutable References

[ ]:
```
fn main() {
    let mut s = String::from("hello");

    change(&mut s);
}

fn change(some_string: &mut String) {
    some_string.push_str(", world");
}
main();
```

Mutable references have one big restriction: you can have **only one mutable reference** to a particular piece of data in a particular scope. This code will fail:

[ ]:
```
fn main() {
    let mut s = String::from("hello");

    let r1 = &mut s;
    let r2 = &mut s;

    println!("{}, {}", r1, r2);
}
main();
```

The benefit of having this restriction is that Rust can prevent *data races* at compile time. A data race is similar to a race condition and happens when these three behaviors occur:

- Two or more pointers access the same data at the same time.
- At least one of the pointers is being used to write to the data.
- There's no mechanism being used to synchronize access to the data.

A similar rule exists for combining mutable and immutable references. We cannot have a mutable reference while we have an immutable one. Users of an immutable reference don't expect the values to suddenly change out from under them! This code results in an error:

[ ]:
```
fn main() {
    let mut s = String::from("hello");

    let r1 = &s; // no problem
    let r2 = &s; // no problem
    let r3 = &mut s; // BIG PROBLEM

    println!("{}, {}, and {}", r1, r2, r3);
}
main();
```

1.2 Dangling pointer

In languages with pointers, it's easy to erroneously create a **dangling pointer** (a pointer that references a location in memory that may have been given to someone else) by freeing some memory while preserving a pointer to that memory. In Rust, by contrast, the compiler guarantees that references will never be dangling references: if you have a reference to some data, the compiler will ensure that the data will not go out of scope before the reference to the data does.

Example of a dangling pointer:

[ ]:
```
fn main() {
    let reference_to_nothing = dangle();
}

fn dangle() -> &String {
    let s = String::from("hello");

    &s // we return a reference to the String, s
} // Here, s goes out of scope, and is dropped. Its memory goes away.
  // Danger!

main();
```

The solution here is to return the String directly:

[ ]:
```
fn no_dangle() -> String {
    let s = String::from("hello");

    s
}
```

2 Validating References with Lifetimes

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. We must annotate types when multiple types are possible. In a similar way, we must annotate lifetimes when the lifetimes of references could be related in a few different ways. Rust requires us to annotate the relationships using generic lifetime parameters to ensure the actual references used at runtime will definitely be valid.

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

[ ]:
```
// the following code will not compile because the lifetime of x ('b) is␣
 ↪shorter than the lifetime of r ('a)
{
    let r;                   // ---------+-- 'a
                             //          |
    {                        //          |
        let x = 5;         // -+-- 'b  |
        r = &x;            //  |       |
    }                      // -+       |
                           //          |
    println!("r: {}", r); //          |
}
```

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

2.1 Lifetime annotation syntax

Lifetime annotations describe the relationships of the lifetimes of multiple references to each other without affecting the lifetimes.

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

2.2 Lifetime annotations in function signatures

[ ]:
```
/* for some lifetime 'a, the function takes two parameters,
   both of which are string slices that live at least as long as lifetime 'a
*/
fn longest<'a>(x: &'a str, y: &'a str) -> &'a str {
    if x.len() > y.len() {
        x
    } else {
        y
    }
}

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

When a function has references to or from code outside that function, it becomes almost impossible for Rust to figure out the lifetimes of the parameters or return values on its own. The lifetimes might be different each time the function is called. This is why we need to annotate the lifetimes manually.

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`.

2.3 Lifetime annotations in struct definitions

It's possible for structs to hold references, but in that case we would need to add a lifetime annotation on every reference in the struct's definition.

[ ]:
```
fn main() {
    #[derive(Debug)]
    // an instance of ImportantExcerpt can't outlive the reference it holds in␣
 ↪its part field.
    struct ImportantExcerpt<'a> {
        part: &'a str,
    }

    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 };
    println!("{:#?}", i);
}
main();
```

2.4 Lifetime annotations in method definitions

The lifetime parameter declaration after `impl` and its use after the type name are required, but we're not required to annotate the lifetime of the reference to self.

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

2.5 The static lifetime

The static lifetime means the reference can live for the entire duration of the program. All string literals have the `'static` lifetime, which we can annotate as follows:

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

3 Exercises

Exercise 1: Identify the errors in the following code and correct them.

[ ]:
```
fn main() {
    let mut x = 5;

    let y = &mut x;

    *y += 1;

    println!("{}", x);
    // expect to print 6
    println!("{}", y);
}
main();
```

Solution:

[ ]:
```
fn main() {
    let mut x = 5;

    {
        let y = &mut x; // -+ &mut borrow starts here
        *y += 1;        //  |
    }                   // -+ ... and ends here

    println!("{}", x);  // <- try to borrow x here
}
main();
```