Rust - Day 5

References and Borrowing

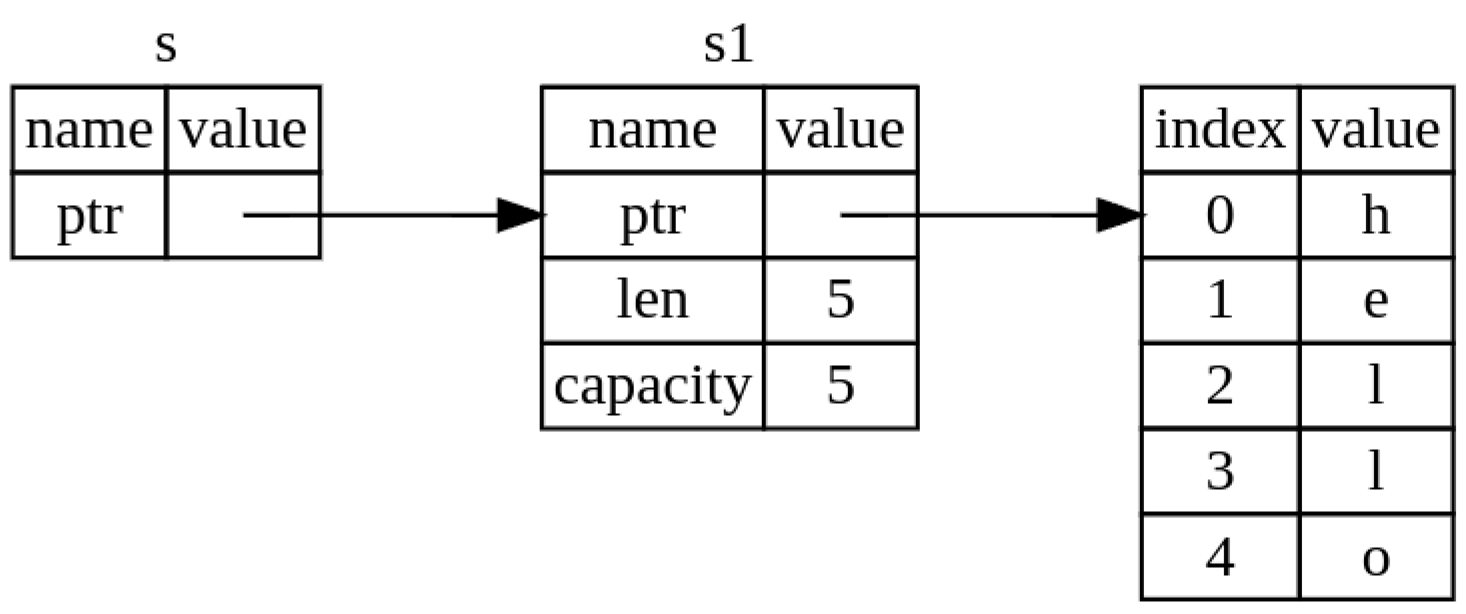
```
fn main() {
    let s1 = String::from("hello");
    let (s2, len) = calculate_length(s1);
    println!("The length of '{s2}' is {len}.");
}

fn calculate_length(s: String) → (String, usize) {
    let length = s.len(); // len() returns the length of a String
    (s, length)
}
```

- The issue with the tuple code in Listing 4-5 is that we have to return the `String` to the calling function so we can still use the `String` after the call to `calculate_length`, because the `String` was moved into `calculate_length`.
- Instead, we can provide a reference to the `String` value.
- A *reference* is like a pointer in that it's an address we can follow to access the data stored at that address; that data is owned by some other variable. Unlike a pointer, a reference is guaranteed to point to a valid value of a particular type for the life of that reference.
- 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:

```
fn main() {
    let s1 = String::from("hello");
    let len = calculate_length(&s1);
    println!("The length of '{s1}' is {len}.");
}
fn calculate_length(s: &String) → usize { // s is a reference to a String
    s.len()
} // Here, s goes out of scope. But because it does not have ownership of what
  // it refers to, it is not dropped.
```

- Tuple code in the variable declaration and the function return value is gone
- **Note that we pass `&s1` into `calculate_length` and, in its definition, we take `&String` rather than `String`.**
- **These ampersands represent *references*, and they allow you to refer to some value without taking ownership of it.**



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

- **The `&s1` syntax lets us create a reference that *refers* to the value of `s1` but does not own it. Because it does not own it, the value it points to will not be dropped when the reference stops being used.**
- Likewise, the signature of the function uses `&` to indicate that the type of the parameter `s` is a reference.
- **The scope in which the variable `s` is valid is the same as any function parameter's scope, but the value is not dropped as `s` does not have the ownership.**
- When functions have references as params instead of actual values, we won't need to return the values as ownership is not transfered.
- **We call the action of creating a reference *borrowing*.**
- **References are immutable by default**

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

    change(&s);
}

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

```
$ cargo run
   Compiling ownership v0.1.0 (file:///projects/ownership)
error[E0596]: cannot borrow `*some_string` as mutable, as it is behind a `&` reference
 --> src/main.rs:8:5
  |
8 |     some_string.push_str(", world");
  |     ^^^^^^^^^^^ `some_string` is a `&` reference, so the data it refers to cannot be borrowed as mutable
  |
help: consider changing this to be a mutable reference
  |
7 | fn change(some_string: &mut String) {
  |                         +++

For more information about this error, try `rustc --explain E0596`.
error: could not compile `ownership` (bin "ownership") due to 1 previous error
```

Mutable References

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

    change(&mut s);
}

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

- We change `s` to be `mut`. Then we create a mutable reference with `&mut s` where we call the `change` function, and update the function to accept a mutable reference with `some_string: &mut String`. This makes it very clear that the `change` function will mutate the value it borrows.
- ❗ Restriction: If you have a mutable reference to a value, you can have no other references to that value. Any code doing the same would fail.

```
    let mut s = String::from("hello");

    let r1 = &mut s;
    let r2 = &mut s;
    println!("{}, {}", r1, r2);
```

```
$ cargo run
   Compiling ownership v0.1.0 (file:///projects/ownership)
error[E0499]: cannot borrow `s` as mutable more than once at a time
 --> src/main.rs:5:14
  |
4 |     let r1 = &mut s;
  |              ------ first mutable borrow occurs here
5 |     let r2 = &mut s;
  |              ^^^^^^ second mutable borrow occurs here
6 |
7 |     println!("{}, {}", r1, r2);
  |                        -- first borrow later used here

For more information about this error, try `rustc --explain E0499`.
error: could not compile `ownership` (bin "ownership") due to 1 previous error
```

- The restriction preventing multiple mutable references to the same data at the same time allows for mutation but in a very controlled fashion.
- 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.**
- Data races cause undefined behavior and can be difficult to diagnose and fix when you're trying to track them down at runtime; Rust prevents this problem by refusing to compile code with data races!
- **But we can still create multiple references under separate scopes like:*

```
    let mut s = String::from("hello");
    {
        let r1 = &mut s;
    } // r1 goes out of scope here, so we can make a new reference with no problems.
    let r2 = &mut s;
```

- Rust enforces a similar rule for combining mutable and immutable references. This code results in an error:

```
    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);
```

```
$ cargo run
   Compiling ownership v0.1.0 (file:///projects/ownership)
error[E0502]: cannot borrow `s` as mutable because it is also borrowed as immutable
 --> src/main.rs:6:14
  |
4 |     let r1 = &s; // no problem
  |              -- immutable borrow occurs here
5 |     let r2 = &s; // no problem
6 |     let r3 = &mut s; // BIG PROBLEM
  |              ^^^^^^ mutable borrow occurs here
7 |
8 |     println!("{}, {}, and {}", r1, r2, r3);
  |                                -- immutable borrow later used here
```

```
For more information about this error, try `rustc --explain E0502`.
error: could not compile `ownership` (bin "ownership") due to 1 previous error
```

- **We *also* cannot have a mutable reference while we have an immutable one to the same value.**
- **Multiple immutable references are allowed because no one who is just reading the data has the ability to affect anyone else's reading of the data.**
- **🌟 Reference's scope starts from where it is introduced and continues through the last time that reference is used**

```
    let mut s = String::from("hello");

    let r1 = &s; // no problem
    let r2 = &s; // no problem
    println!("{r1} and {r2}");
    // variables r1 and r2 will not be used after this point

    let r3 = &mut s; // no problem
    println!("{r3}");
```

Dangling References

- 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.**

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

fn dangle() -> &String { // dangle returns a reference to a String
    let s = String::from("hello"); // s is a new String
    &s // we return a reference to the String, s
} // Here, s goes out of scope, and is dropped. Its memory goes away.
  // Danger!
```

```
$ cargo run
   Compiling ownership v0.1.0 (file:///projects/ownership)
error[E0106]: missing lifetime specifier
 --> src/main.rs:5:16
  |
5 | fn dangle() -> &String {
  |                ^ expected named lifetime parameter
  |
  = help: this function's return type contains a borrowed value, but there is no value for it to be borrowed from
help: consider using the `'static` lifetime, but this is uncommon unless you're returning a borrowed value from a
`const` or a `static`
  |
5 | fn dangle() -> &'static String {
  |                 +++++++
help: instead, you are more likely to want to return an owned value
  |
5 - fn dangle() -> &String {
5 + fn dangle() -> String {
  |

error[E0515]: cannot return reference to local variable `s`
 --> src/main.rs:8:5
```

```
  |
8 |     &s
  |     ^^ returns a reference to data owned by the current function

Some errors have detailed explanations: E0106, E0515.
For more information about an error, try `rustc --explain E0106`.
error: could not compile `ownership` (bin "ownership") due to 2 previous errors
```

- Because `s` is created inside `dangle`, when the code of `dangle` is finished, `s` will be deallocated. But we tried to return a reference to it.
- That means this reference would be pointing to an invalid `String`.
- Solution is to return the `String` directly:

```
fn no_dangle() → String {
    let s = String::from("hello");
    s
}
```

- This works without any problems. Ownership is moved out, and nothing is deallocated.

Rules of References

- At any given time, you can have *either* one mutable reference *or* any number of immutable references.
- References must always be valid.

Slice Type

- *Slices* let you reference a contiguous sequence of elements in a `collection` rather than the whole collection. A slice is a kind of reference, so it does not have ownership.

> Write a function that takes a string of words separated by spaces and returns the first word it finds in that string. If the function doesn't find a space in the string, the whole string must be one word, so the entire string should be returned.

```
fn first_word(s: &String) → ?
```

- The function `first_word` takes an immutable reference of a `String` as parameter.
- What should we return? We don't really have a way to talk about *part* of a string. However, we could return the index of the end of the word, indicated by a space.

```
fn first_word(s: &String) → usize {
    let bytes = s.as_bytes();
    for (i, &item) in bytes.iter().enumerate() {
        if item == b' ' {
            return i;
        }
    }
    s.len()
}
```

- Because we need to go through the `String` element by element and check whether a value is a space, we'll convert our `String` to an array of bytes using the `as_bytes` method.

```
    let bytes = s.as_bytes();
```

- Next, we create an iterator over the array of bytes using the `iter` method:

```
    for (i, &item) in bytes.iter().enumerate() {
```

- **`iter` is a method that returns each element in a collection and that `enumerate` wraps the result of `iter` and returns each element as part of a tuple instead.**
- **The first element of the tuple returned from `enumerate` is the index, and the second element is a reference to the element.**
- This is a bit more convenient than calculating the index ourselves.
- Because the `enumerate` method returns a tuple, we can use patterns to destructure that tuple.
- **In the `for` loop, we specify a pattern that has `i` for the index in the tuple and `&item` for the single byte in the tuple. Because we get a reference to the element from `.iter().enumerate()`, we use `&` in the pattern.**
- **Inside the `for` loop, we search for the byte that represents the space by using the byte literal syntax. If we find a space, we return the position. Otherwise, we return the length of the string by using `s.len()`.**

```
        if item == b' ' {
            return i;
        }
    }
    s.len()
```

- We now have a way to find out the index of the end of the first word in the string, but there's a **problem**.
- We're returning a `usize` which is a separate value not connected to the state of `String` and there's no guarantee that it will still be valid in the future like in this case:

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

    let word = first_word(&s); // word will get the value 5

    s.clear(); // this empties the String, making it equal to ""

    // word still has the value 5 here, but there's no more string that
    // we could meaningfully use the value 5 with. word is now totally invalid!
}
```

- ❗ This program compiles without any errors and would also do so if we used `word` after calling `s.clear()`. Because `word` isn't connected to the state of `s` at all, `word` still contains the value `5`. We could use that value `5` with the variable `s` to try to extract the first word out, but this would be a bug because the contents of `s` have changed since we saved `5` in `word`.
- Luckily, Rust has a solution to this problem: **string slices.**

String Slices

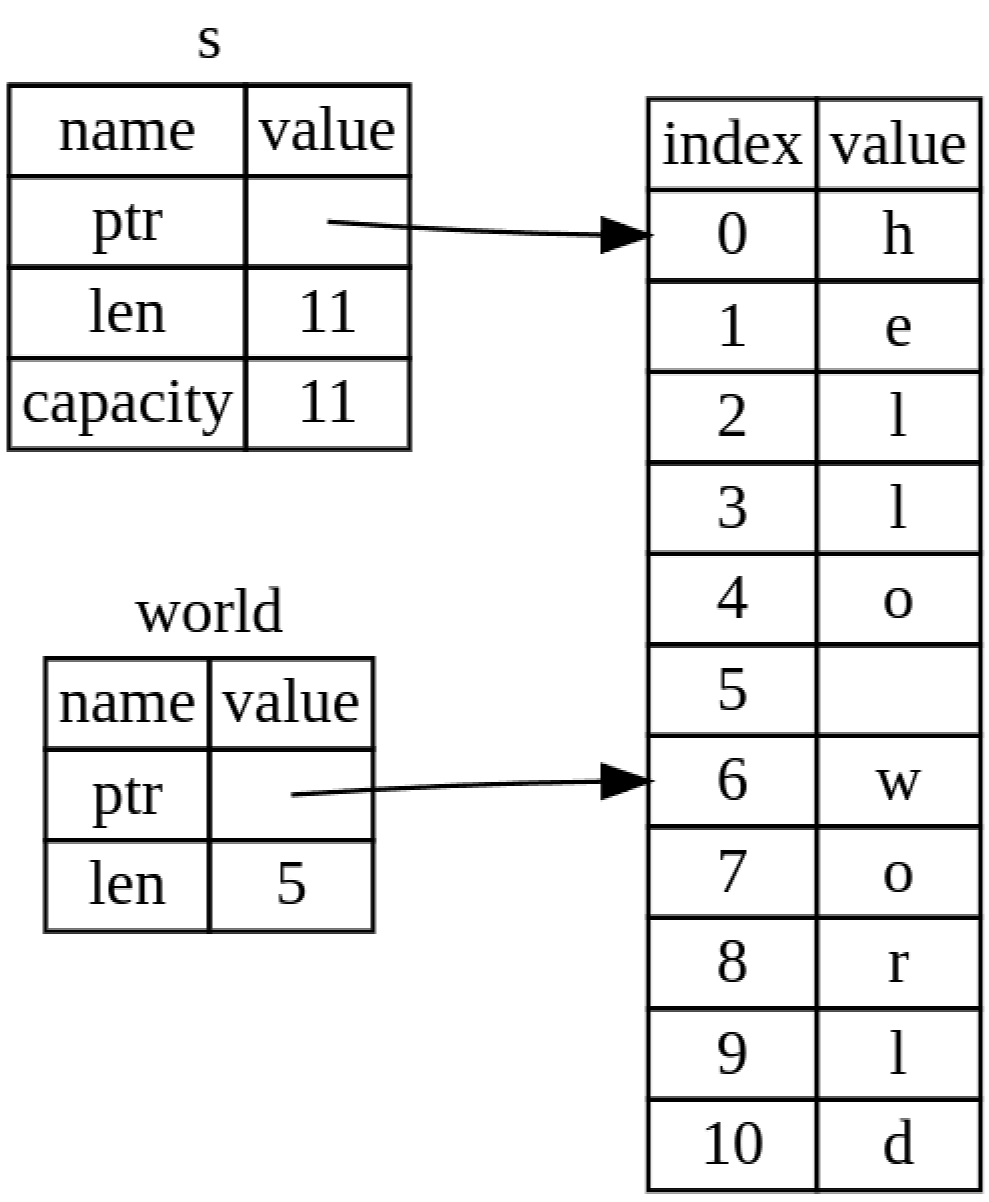
- A *string slice* is a reference to part of a `String`, and it looks like this:

```
    let s = String::from("hello world");

    let hello = &s[0..5];
    let world = &s[6..11];
```

- Rather than a reference to the entire `String`, `hello` is a reference to a portion of the `String`, specified in the extra `[0..5]` bit.
- **We create slices using a range within brackets by specifying `[starting_index..ending_index]`, where `starting_index` is the first position in the slice and `ending_index` is one more than the last position in the slice.**
- Internally, the slice data structure stores the starting position and the length of the slice, which corresponds to `ending_index` minus `starting_index`.

- **So, in the case of `let world = &s[6..11];`, `world` would be a slice that contains a pointer to the byte at index 6 of `s` with a length value of `5`.**



Code snippets for .. range syntax

- **Get a slice from start to nth index**. Here both `slice1` and `slice2` are the same.

```
let s = String::from("hello");

let slice1 = &s[0..2]
let slice2 = &s[..2]
```

- **Get a slice from nth index to last index**

```
let s = String::from("hello");

let len = s.len();

let slice1 = &s[3..len];
let slice2 = &s[3..];
```

- **Get the slice of the complete string**

```
let s = String::from("hello");

let len = s.len();

let slice = &s[0..len];
let slice = &s[..];
```

> Note: String slice range indices must occur at valid UTF-8 character boundaries. If you attempt to create a string slice in the middle of a multibyte character, your program will exit with an error.

- With all this information in mind, let's rewrite `first_word` to return a slice. The type that signifies "string slice" is written as `&str`:

```
fn first_word(s: &String) -> &str {
    let bytes = s.as_bytes();

    for (i, &item) in bytes.iter().enumerate() {
        if item == b' ' {
            return &s[0..i];
        }
    }
    &s[..]
}
```

- Using the slice version of `first_word` will throw a compile-time error:

```
fn main() {
    let mut s = String::from("hello world");
    let word = first_word(&s);

    s.clear(); // error!

    println!("the first word is: {word}");
}
```

```
$ cargo run
   Compiling ownership v0.1.0 (file:///projects/ownership)
error[E0502]: cannot borrow `s` as mutable because it is also borrowed as immutable
  --> src/main.rs:18:5
   |
16 |     let word = first_word(&s);
   |                           -- immutable borrow occurs here
17 |
18 |     s.clear(); // error!
   |     ^^^^^^^^^ mutable borrow occurs here
19 |
20 |     println!("the first word is: {word}");
   |                                  ------ immutable borrow later used here

For more information about this error, try `rustc --explain E0502`.
error: could not compile `ownership` (bin "ownership") due to 1 previous error
```

- Recall from the borrowing rules that if we have an immutable reference to something, we cannot also take a mutable reference.
- **Because `clear` needs to truncate the `String`, it needs to get a mutable reference.**
- **The `println!` after the call to `clear` uses the reference in `word`, so the immutable reference must still be active at that point.**
- **Rust disallows the mutable reference in `clear` and the immutable reference in `word` from existing at the same time, and compilation fails.**

String literals as Slices

- Recall that we talked about string literals being stored inside the binary. Now that we know about slices, we can properly understand string literals:

```
let s = "Hello, world!";
```

- The type of `s` here is `&str`: it’s a slice pointing to that specific point of the binary. This is also why string literals are immutable; `&str` is an immutable reference.

String Slices as Parameters

- Knowing that you can take slices of literals and `String` values leads us to one more improvement on `first_word`, and that's its signature:

```
fn first_word(s: &String) → &str {
```

- A more experienced Rustacean would write the signature shown in Listing 4-9 instead because it allows us to use the same function on both `&String` values and `&str` values.

```
fn first_word(s: &str) → &str {
```

- If we have a string slice, we can pass that directly. If we have a `String`, we can pass a slice of the `String` or a reference to the `String`. This flexibility takes advantage of *deref coercions*, a feature we will cover in the "Implicit Deref Coercions with Functions and Methods" section of Chapter 15.
- Defining a function to take a string slice instead of a reference to a `String` makes our API more general and useful without losing any functionality:

```
fn main() {
    let my_string = String::from("hello world");

    // `first_word` works on slices of `String`s, whether partial or whole
    let word = first_word(&my_string[0..6]);
    let word = first_word(&my_string[..]);
    // `first_word` also works on references to `String`s, which are equivalent
    // to whole slices of `String`s
    let word = first_word(&my_string);

    let my_string_literal = "hello world";

    // `first_word` works on slices of string literals, whether partial or whole
    let word = first_word(&my_string_literal[0..6]);
    let word = first_word(&my_string_literal[..]);

    // Because string literals *are* string slices already,
    // this works too, without the slice syntax!
    let word = first_word(my_string_literal);
}
```

Other Slices

- String slices, as you might imagine, are specific to strings. But there's a more general slice type too. Consider this array:

```
let a = [1, 2, 3, 4, 5];
```

- Just as we might want to refer to part of a string, we might want to refer to part of an array. We'd do so like this:

```
let a = [1, 2, 3, 4, 5];

let slice = &a[1..3];

assert_eq!(slice, &[2, 3]);
```

- This slice has the type `&[i32]`. It works the same way as string slices do, by storing a reference to the first element and a length. You'll use this kind of slice for all sorts of other collections.

Summary

- **The concepts of ownership, borrowing, and slices ensure memory safety in Rust programs at compile time.**
- **The Rust language gives you control over your memory usage in the same way as other systems programming languages, but having the owner of data automatically clean up that data when the owner goes out of scope means you don’t have to write and debug extra code to get this control.**