한 번에 끝내는 블록체인 개발 A to Z

Chapter 1

Rust Introduction

Chapter 1 Rust Introduction

What is Ownership?

Ownership

Ownership is Rust's most unique feature and has deep implications for the rest of the language. It enables Rust to make memory safety guarantees without needing a garbage collector, so it's important to understand how ownership works.

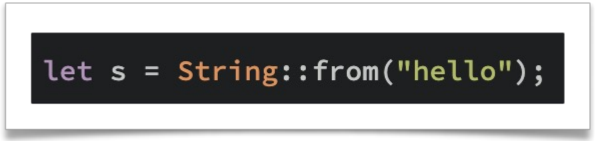
	Stack	Heap
Speed	Fast	Slow
Data Size	A Known Fixed	Dynamic
Access Order	Last in First out	No Matter
Cleared When	Function Finished	Ownership!!

Ownership Rules

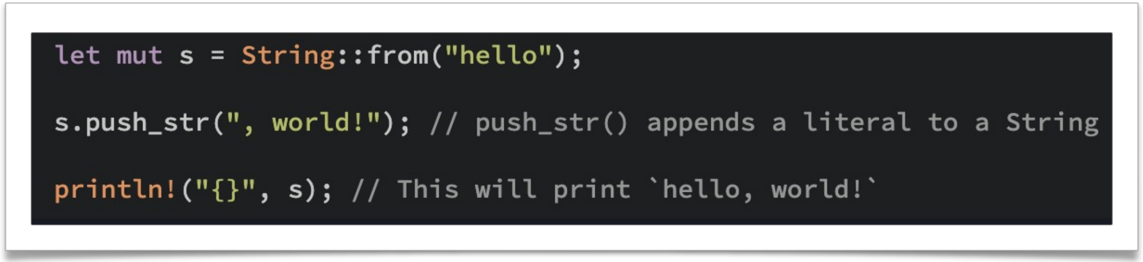
- Each value in Rust has an owner.
- There can only be one owner at a time.
- When the owner goes out of scope, the value will be dropped.

String Type

We want to look at data that is stored on the heap and explore how Rust knows when to clean up that data, and the String type is a great example.



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



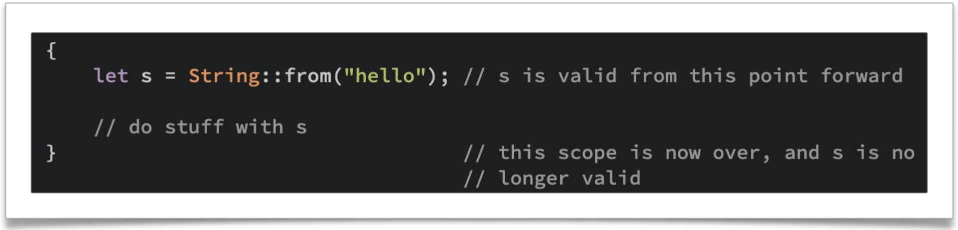
```
let mut s = String::from("hello");

s.push_str(", world!"); // push_str() appends a literal to a String

println!("{}", s); // This will print `hello, world!`
```

Allocation

In languages with a garbage collector (GC), the GC keeps track of and cleans up memory that isn’t being used anymore, and we don’t need to think about it.

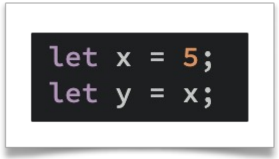
In most languages without a GC, it’s our responsibility to identify when memory is no longer being used and call code to explicitly free it, just as we did to request it. Doing this correctly has historically been a difficult programming problem.

Rust takes a different path: the memory is automatically returned once the variable that owns it goes out of scope.



```
{
    let s = String::from("hello"); // s is valid from this point forward

    // do stuff with s
}                                  // this scope is now over, and s is no
                                   // longer valid
```

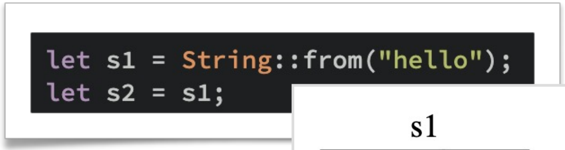
Move



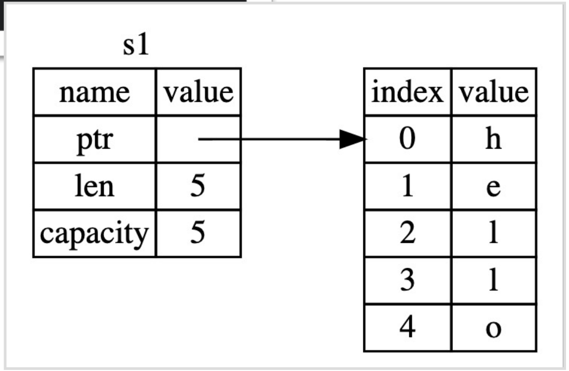
```
let x = 5;
let y = x;
```

“bind the value 5 to x; then make a copy of the value in x and bind it to y.”

Integers are simple values with a known, fixed size, and these two 5 values are pushed onto the stack.



```
let s1 = String::from("hello");
let s2 = s1;
```



s1

name	value
ptr	
len	5
capacity	5

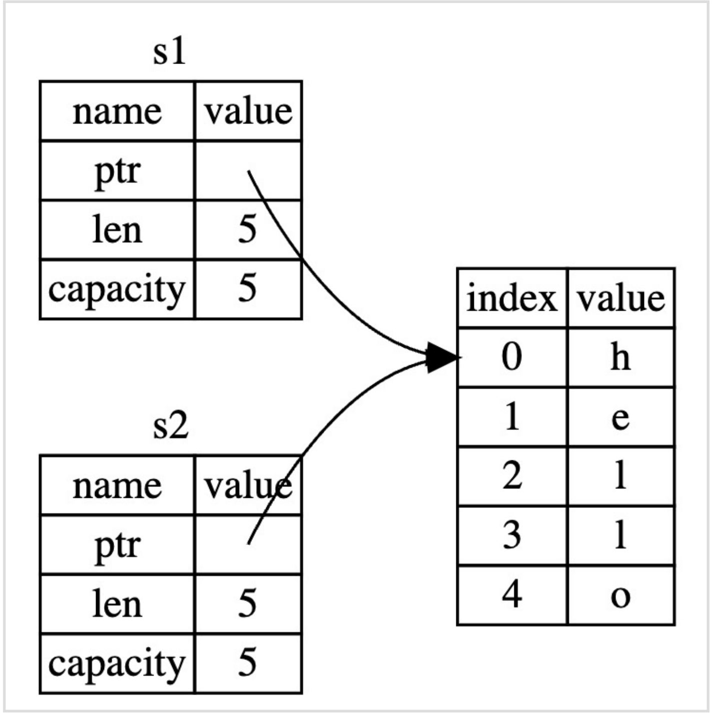
index	value
0	h
1	e
2	l
3	l
4	o

A String is made up of three parts, shown on the left: a pointer to the memory that holds the contents of the string, a length, and a capacity. This group of data is stored on the stack. On the right is the memory on the heap that holds the contents.

Move

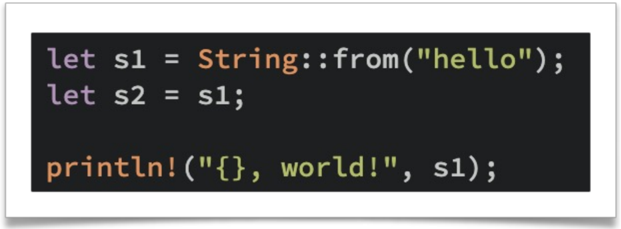
When we assign s1 to s2, the String data is copied, meaning we copy the pointer, the length, and the capacity that are on the stack. We do not copy the data on the heap that the pointer refers to.

Rust automatically calls the drop function and cleans up the heap memory for that variable. This is a problem:
when s2 and s1 go out of scope, they will both try to free the same memory. This is known as a double free error and is one of the memory safety bugs we mentioned previously.

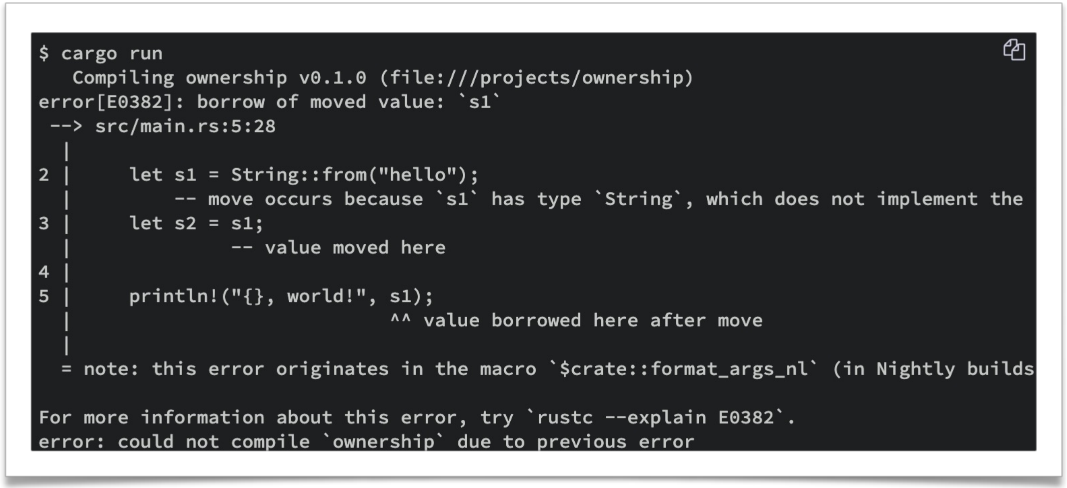


Move

To ensure memory safety, after the line let s2 = s1, Rust considers s1 as no longer valid. Therefore, Rust doesn't need to free anything when s1 goes out of scope. Check out what happens when you try to use s1 after s2 is created; it won't work:



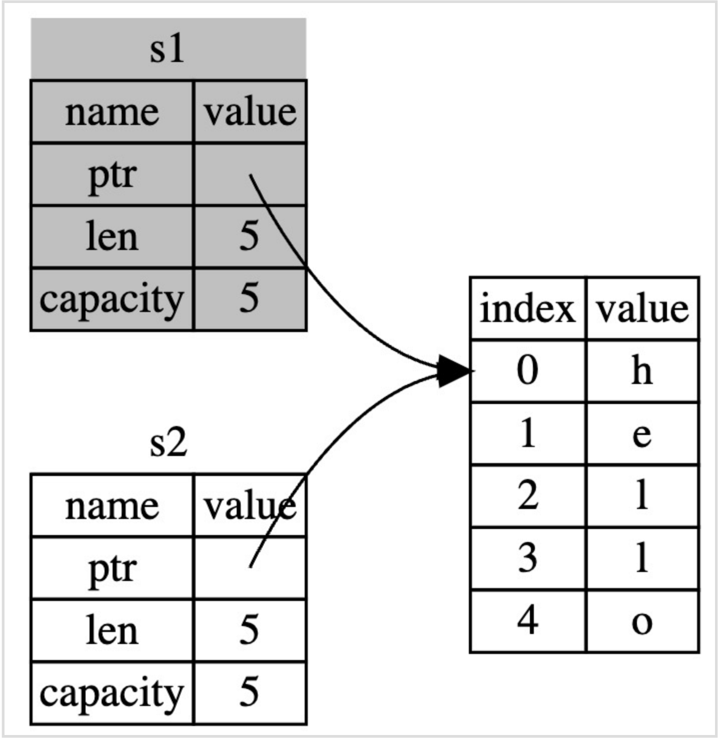
```rust
let s1 = String::from("hello");
let s2 = s1;

println!("{}, world!", s1);
```



```
$ cargo run
   Compiling ownership v0.1.0 (file:///projects/ownership)
error[E0382]: borrow of moved value: `s1`
 --> src/main.rs:5:28
  |
2 |     let s1 = String::from("hello");
  |         -- move occurs because `s1` has type `String`, which does not implement the
3 |     let s2 = s1;
  |              -- value moved here
4 |
5 |     println!("{}, world!", s1);
  |                            ^^ value borrowed here after move
  |
  = note: this error originates in the macro `$crate::format_args_nl` (in Nightly builds

For more information about this error, try `rustc --explain E0382`.
error: could not compile `ownership` due to previous error
```

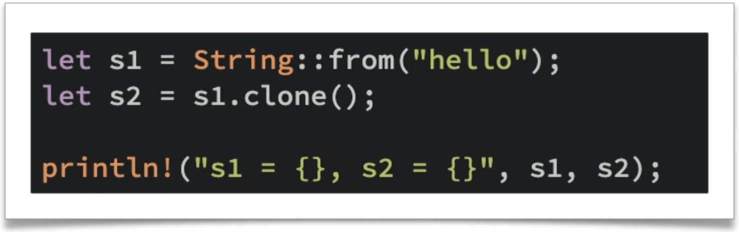
Move

If you’ve heard the terms shallow copy and deep copy while working with other languages, the concept of copying the pointer, length, and capacity without copying the data probably sounds like making a shallow copy. But because Rust also invalidates the first variable, instead of calling it a shallow copy, it’s known as a move.

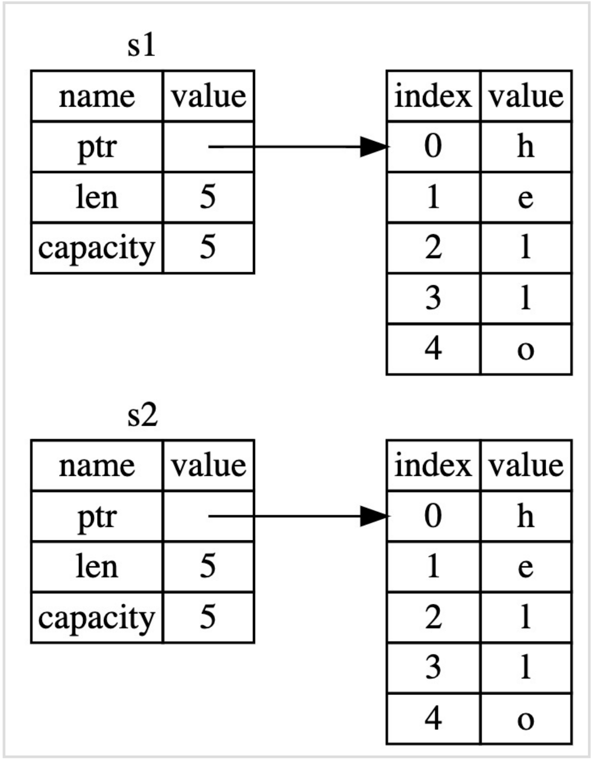


Clone

If we do want to deeply copy the heap data of the String, not just the stack data, we can use a common method called clone.



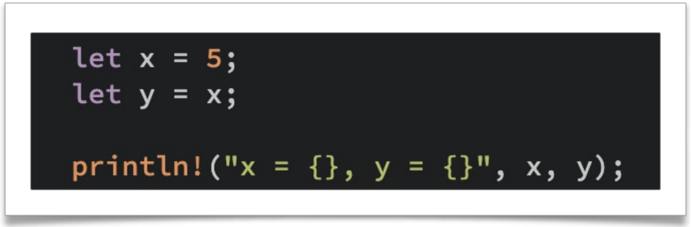
```
let s1 = String::from("hello");
let s2 = s1.clone();

println!("s1 = {}, s2 = {}", s1, s2);
```



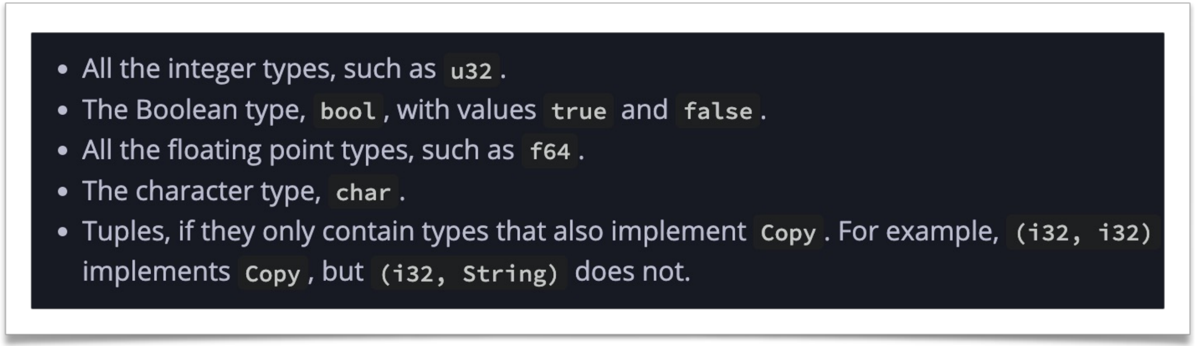
Copy

We don’t have a call to clone, but x is still valid and wasn’t moved into y.

The reason is that types such as integers that have a known size at compile time are stored entirely on the stack, so copies of the actual values are quick to make. That means there’s no reason we would want to prevent x from being valid after we create the variable y.



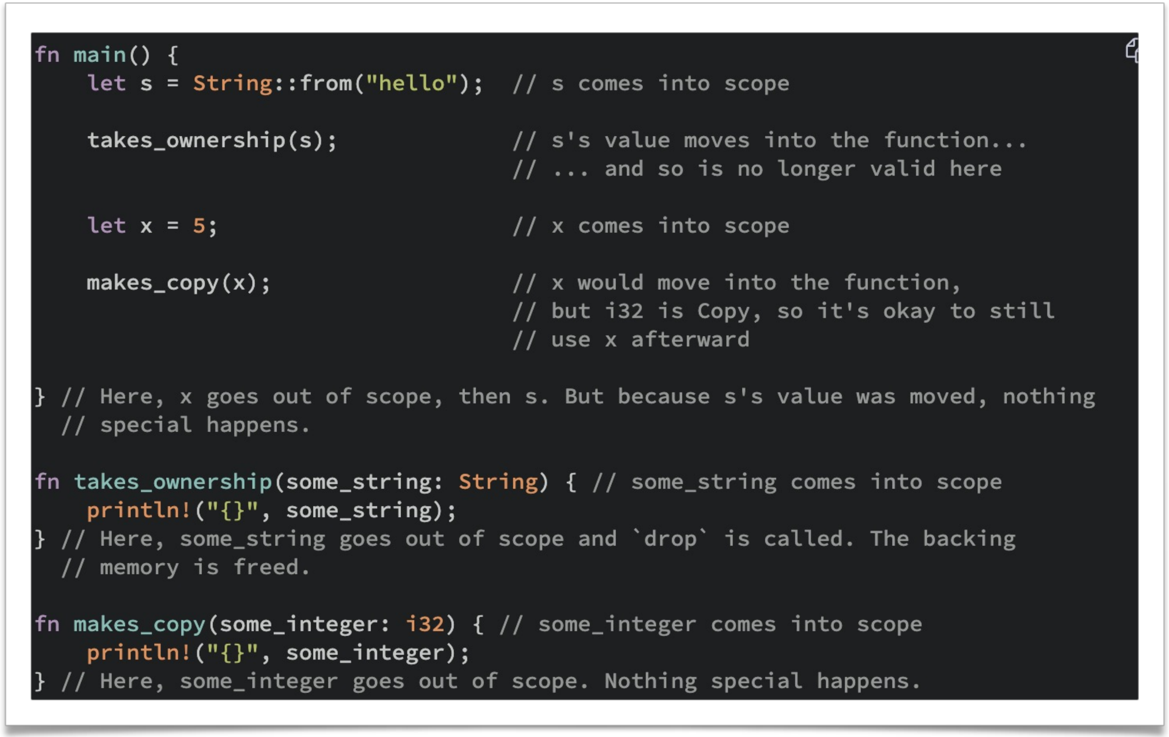
```
let x = 5;
let y = x;

println!("x = {}, y = {}", x, y);
```



- All the integer types, such as `u32`.
- The Boolean type, `bool`, with values `true` and `false`.
- All the floating point types, such as `f64`.
- The character type, `char`.
- Tuples, if they only contain types that also implement `Copy`. For example, `(i32, i32)` implements `Copy`, but `(i32, String)` does not.

Ownership Functions

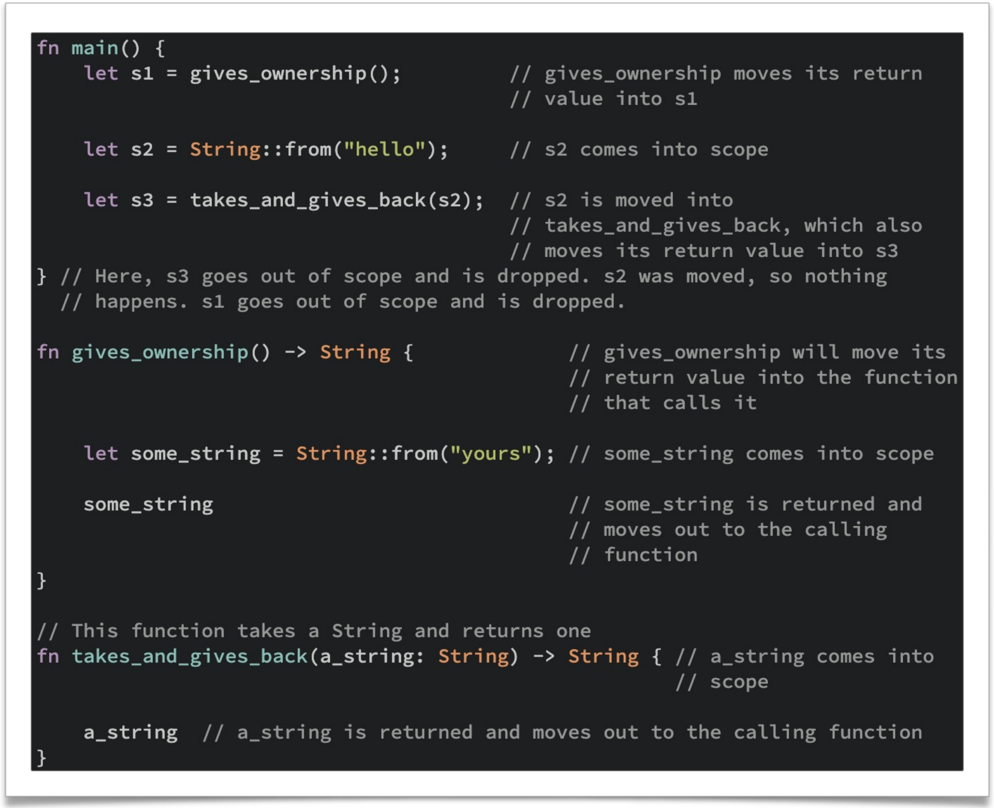
The mechanics of passing a value to a function are similar to those when assigning a value to a variable. Passing a variable to a function will move or copy, just as assignment does.



```
fn main() {
    let s = String::from("hello");  // s comes into scope

    takes_ownership(s);             // s's value moves into the function...
                                    // ... and so is no longer valid here

    let x = 5;                      // x comes into scope

    makes_copy(x);                  // x would move into the function,
                                    // but i32 is Copy, so it's okay to still
                                    // use x afterward

} // Here, x goes out of scope, then s. But because s's value was moved, nothing
  // special happens.

fn takes_ownership(some_string: String) { // some_string comes into scope
    println!("{}", some_string);
} // Here, some_string goes out of scope and `drop` is called. The backing
  // memory is freed.

fn makes_copy(some_integer: i32) { // some_integer comes into scope
    println!("{}", some_integer);
} // Here, some_integer goes out of scope. Nothing special happens.
```

Return Values

Returning values can also transfer ownership.



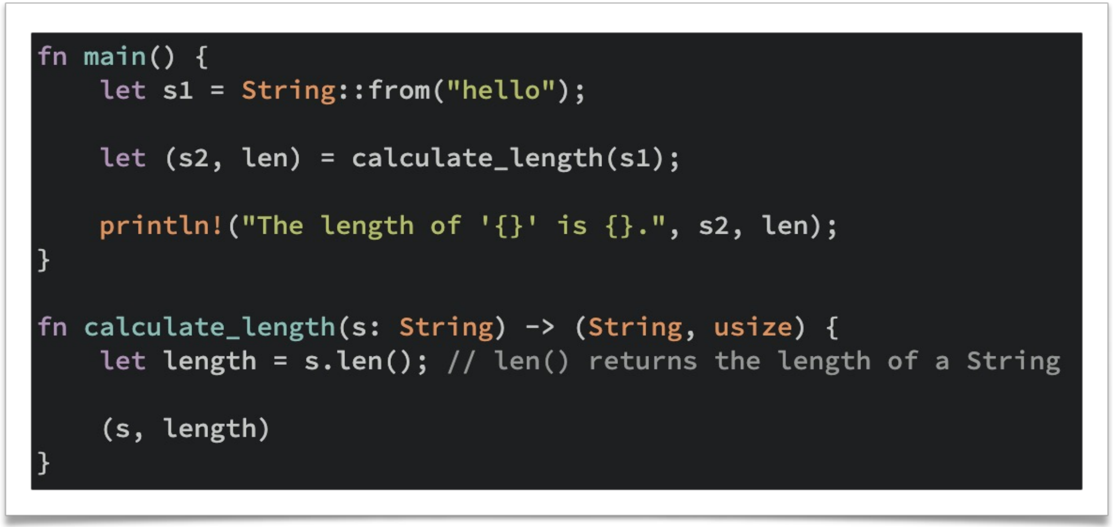
```rust
fn main() {
    let s1 = gives_ownership();         // gives_ownership moves its return
                                        // value into s1

    let s2 = String::from("hello");     // s2 comes into scope

    let s3 = takes_and_gives_back(s2);  // s2 is moved into
                                        // takes_and_gives_back, which also
                                        // moves its return value into s3
} // Here, s3 goes out of scope and is dropped. s2 was moved, so nothing
  // happens. s1 goes out of scope and is dropped.

fn gives_ownership() -> String {             // gives_ownership will move its
                                             // return value into the function
                                             // that calls it

    let some_string = String::from("yours"); // some_string comes into scope

    some_string                              // some_string is returned and
                                             // moves out to the calling
                                             // function
}

// This function takes a String and returns one
fn takes_and_gives_back(a_string: String) -> String { // a_string comes into
                                                      // scope

    a_string  // a_string is returned and moves out to the calling function
}
```

Return Values

What if we want to let a function use a value but not take ownership?

It’s quite annoying that anything we pass in also needs to be passed back if we want to use it again, in addition to any data resulting from the body of the function that we might want to return as well.

→ references



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

    let (s2, len) = calculate_length(s1);

    println!("The length of '{}' is {}.", s2, len);
}

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

    (s, length)
}
```