**Rust Ownership — Explained for Beginners**

[[](https://medium.com/@vennilapugazhenthi?source=post_page-----de70de16b099--------------------------------)](https://medium.com/@vennilapugazhenthi?source=post_page-----de70de16b099--------------------------------)

[Vennila Pugazhenthi](https://medium.com/@vennilapugazhenthi?source=post_page-----de70de16b099--------------------------------)

·

Follow

5 min read

·

Jul 30

22

1

Memory management in Rust is done through a set of rules known as *Ownership*.

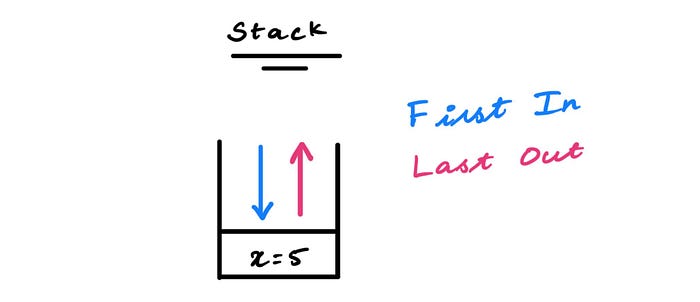
**Ownership Rules:**

* 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.

**Types of Memory:**

Since Rust is a systems programming language, whether a value is present in a stack or heap affects how the code is written.

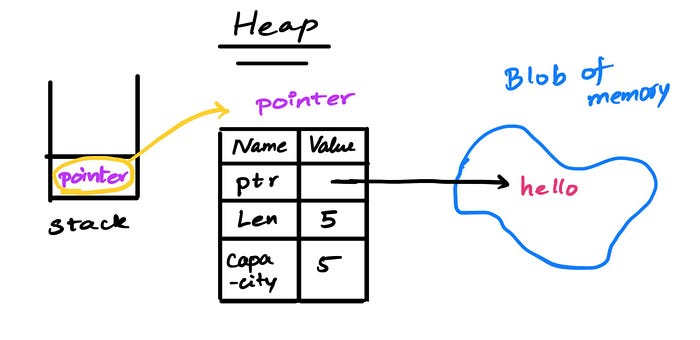
**Stack:**



Representation of value getting pushed into a stack

All data whose size is fixed and known at compile time will be stored in the stack. For example, an integer variable ‘x’ has a value of 5. Its size is fixed and known at compile time, so it gets stored in the stack.

**Heap:**



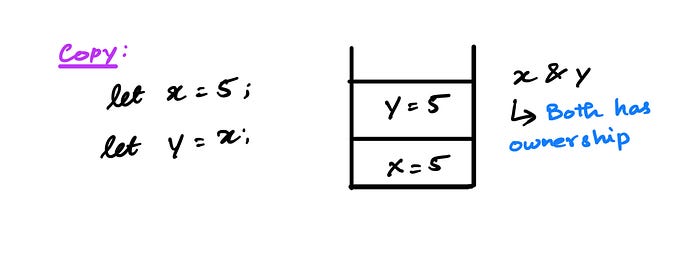
Representation of data in the heap

The data whose size is unknown at compile time or if it might change gets stored in the heap. The memory allocator finds a big enough empty spot to store the data and returns a pointer. The pointer contains the memory address of the data.

**Ownership (Stack):**

let x = 5;  
let y = x;

In this example, the data is of integer type. So, the data reside in a stack. When ‘y’ is initialized with ‘x’, the data get **copied** to ‘y’. However, the ownership of ‘x’ doesn’t change, and ‘y’ gets ownership of a new copy of ‘5’.



Representation of stack data getting copied

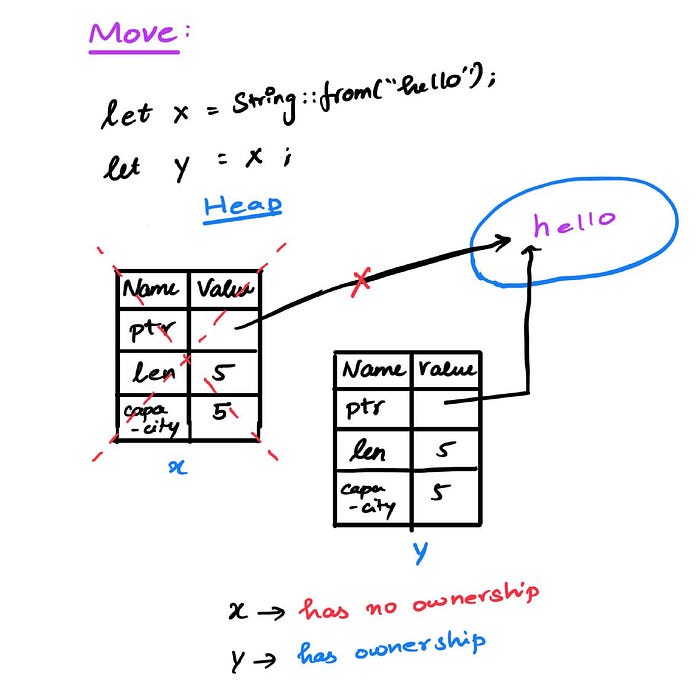
Rule: All the scalar data types in Rust have the ‘Copy’ trait implemented, which will have the same behavior as above. Some scalar datatypes with the ‘Copy’ trait are Integer, Boolean, Floating-point, Character, and Tuples if they only contain types that also implement ‘Copy’.

**Ownership (Heap):**

**Move:**

let x = String::from("hello");  
 let y = x;

In this example, ‘x’ is of String type and it’s stored in the heap. Initializing a variable with another variable with data in the heap will result in data **moving** to a new variable. Here, “hello” will move to ‘y’, and ‘y’ will gain ownership. ‘x’ will lose ownership and we can no longer use it in the subsequent code.



Representation of data move in the heap

let x = String::from("hello");  
 let y = x;  
  
 println!("{}, world!", x); // throws an error

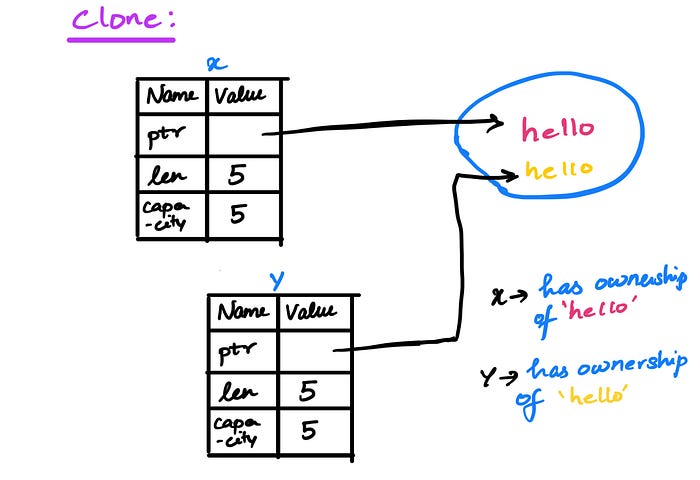
If you try to use ‘x’ after it lost its ownership, then the program will give you an error.

**Clone:**

If you want to make a copy of the value of ‘x’ without ‘x’ losing its ownership, you can use ‘clone()’. This will make a deep copy of the data, which means both the data in the heap and the pointer will be replicated.

let x = String::from("hello");  
 let y = s1.clone();  
  
 println!("x = {}, y = {}", x, y);

Here, both the ‘x’ & ‘y’ will have ownership over their own copy of data.



Representation of data clone in the heap

**Ownership in Functions:**

So far, we went through three concepts:

1. For data in stack, the value gets **copied** and doesn’t lose it ownership.

2. For data in the heap, the value gets **moved** and it loses ownership.

3. For data in heap and ‘clone()’ is used, then the value gets **cloned**and it doesn’t lose its ownership.

Now, let's see how these concepts come into action when used in a function.

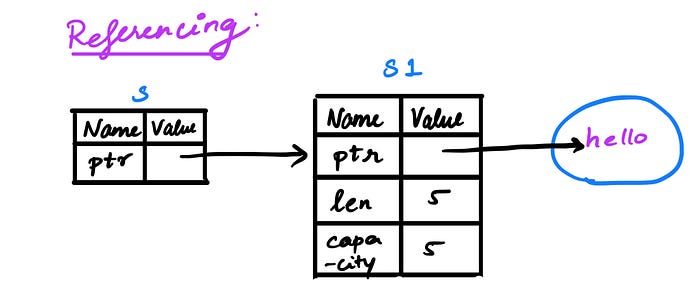
fn main() {  
 let s = String::from("hello"); // s comes into scope  
  
 takes\_ownership(s); // s's value moves into the function...  
 // ... and so is no longer valid here  
  
}  
  
fn takes\_ownership(some\_string: String) { // some\_string comes into scope  
 println!("{}", some\_string);  
} // Here, some\_string goes out of scope and `drop` is called. The backing  
 // memory is freed.

Here since ‘s’ is of type “String” and is stored in the heap, the value **moves,** and ‘s’ loses its ownership. If ‘s’ was of type Integer, then the value would have been **copied,**and ‘s’ won’t have lost its ownership.

**References:**

If you want to use a value without dealing with ownership, you can create a reference.

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()  
}



Representation of reference of data in the heap

Here, ‘s’ doesn’t have ownership of “hello” but it can access the value. The process of creating a reference is known as borrowing. By default, references are immutable, and ‘s’ cannot modify the value “hello”.

Another point to note is that the reference ‘s’ is only valid as long as ‘s1’ is valid. When ‘s1’ goes out of the scope and it loses its ownership, you cannot create a reference for it.

**Mutable References:**

If you want to modify the value that the reference is pointing to, then you need to explicitly add ‘mut’ in the variable declaration. With mutable references, you can modify or edit the value without having ownership of it.

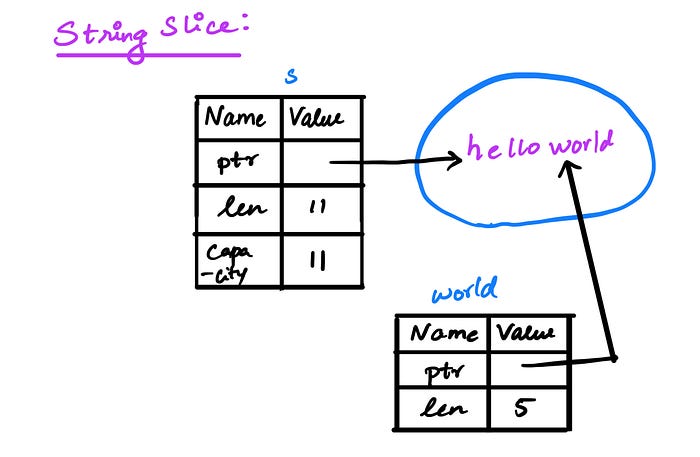
fn main() {  
 let mut s = String::from("hello");  
  
 change(&mut s);  
}  
  
fn change(some\_string: &mut String) {  
 some\_string.push\_str(", world");  
}

There’s one big restriction with using mutable reference: if you have a mutable reference of a value, then you cannot any other reference in the same scope. This doesn’t mean that you cannot have a mutable reference and an immutable reference of the same value within the program. Just that, it shouldn’t be within the same scope of the program like within the same function. This condition helps to prevent any data races.

**String Slices:**

A string slice is a reference that points to part of a String without no ownership of the data.

let s = String::from("hello world");  
  
 let world = &s[6..11];



Representation of string slice

The String Slice is of type ‘&str’ which is different from ‘&String’. The difference is ‘&str’ is a reference to part of the String while ‘&String’ is a reference to the whole value of the String.

As string slice points to a part of the string data, it’s an immutable reference and it can never be a mutable reference.

String slice is specific to String datatype, but you can have a slice of other collections. For example, below is a slice of an integer array of type ‘&[i32]’

let a = [1, 2, 3, 4, 5];  
  
let slice = &a[1..3];

**Conclusion:**

Understanding how ownership works differently depending on whether data is on stack or heap and understanding how references work is crucial for writing code in Rust. Hope this article gave you a foundation for understanding how memory works in Rust.