What is Ownership?

Before we discuss what is ownership, let's talk about

The Stack & The Heap

- Memory the program can use during runtime.

- Stack:
  - LIFO.
  - All data stored here must have a known fixed size.
- Heap:
  - Process requests a certain amount of space and OS finds a spot that is big enough and returns a pointer (i.e. allocating on the heap).

- Analogies:
  - Stack of plates: Stack
  - Arriving at a restaurant and requesting a spot : Heap.

- Access time:
  - Pushing to stack is faster than allocating to the heap (i.e. no need to search for empty spot).
  - Accessing data on the stack is slower than accessing data on the heap (i.e. no need to follow a pointer).

- When we call a function, the values passed into the function and the function's local variables get pushed onto the stack, when done they get popped off.

Ownership rules

- Each value in Rust has a variable that's called its *owner*.
- There can only be one owner at a time.
- When the owner goes out of scope (range where an item is valid), the value will be dropped.

The `String` Type

- `String` s are stored on the heap rather than the other data types studied previously, which stored on the stack (and popped-off when scope is over).
- We'll use this fact to explore how Rust knows when to clean up the heap data.

- Two types:
  - `str` : string literal.
    - cannot be mutated (concatenate, etc.).
    - must be hardcoded.
    - stored in the program binary.

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

  - `String` .
    - syntax:

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

    - can be mutated.
      - mutating:

      ```
      let s = String::from("hello");
      s.push_str(", world!");
      println!("{}", s);
      ```

    - doesn't need to be hardcoded (io).
    - stored in the heap.

    > Recall `::` allows us to grab the `from` function from `String` .

- To utilize the heap we must:
  - Request memory from OS at runtime.
    - This is done by `String::from()`
  - Return memory to OS when we're done.
    - Languages with a garbage collector (Java) don't need to worry about this step.
    - Languages without a garbage collector make it our responsibility to identify when memory is no longer being used and call code to explicitly return it (i.e. `allocate` -> `free`).
    - In Rust, memory is automatically returned once the variable goes out of scope.

      ```
      {
        let s = String::from("hello");
      } // s is no longer valid.
      ```

- Rust calls the `drop` fxn after the variable in question goes out of scope.

Ways variables and Data Interact: Move

```
let x = 5; // bind 5 to x
let y = x; // create a copy of x and bind to y
```

- This is possible because these are fixed size variables stored in the stack.

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

- In this case, we are :
  1. Allocating s1 (which constitutes of a pointer, length, and capacity (stored in the stack))) to the heap;
  2. Setting s2's pointer to equal s1.
- We are **not** copying the heap data.

- Now, when s1 and s2 go out of scope, they will both try to `drop` the same heap location. This is called the **double free error**.
- To avoid this, when we set `s2 = s1` , Rust actually considers the value of s1 to no longer be valid. Meaning:

```
let s1 = String::from("hello");
let s2 = s1; // this is called a "move"
println!("{}", s1);
```

> Generates an error.

Ways variables and Data Interact: Clone

- What if we do want to copy the data from one string to another and maintain both?

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

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

- In this case, the heap data is being copied and this operation is very expensive on runtime performance.

> Stack-stored variables don't need cloning. In fact, they have a `Copy` trait which means their stack values can be just copied onto another variable. These data types have the `Copy` trait:
>
> > integers, bool, floats, char, tuples (if of the previously mentioned data types).

Ownership and Functions

- Passing a variable to a function will move or copy just as assignment does.

```
fn main(){
  let s = String::from("ownership is cool!"); // s comes into scope
  foo_string(s); //s is moved to foo_string()
  // s is no longer valid here.

  let i = 5; // i comes into scope
  foo_int(i); // i is copied to foo_int()
} // i is popped off stack

fn foo_string(s1 : String){ // s1 comes into scope
  println!("{}", s);
} // rust calls drop and heap of s1 is cleaned

fn foo_int(i1 : i32){ // i1 comes into scope
  println!("{}", i);
} // i1 is popped off stack
```

Return Values and Scope

- Return values can also give ownership.

```
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 goes out of scope but 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("hello"); // some_string comes into scope

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

// takes_and_gives_back will take a String and return 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
}
```

- Rust also has a feature for the cases where we don't want to give up ownership when passing the `String` value onto the function.