Ownership and Functions

This lesson discusses how ownership works by using functions.

We'll cover the following

As discussed in the previous lesson, the assignment of a variable to another variable will copy or move it. In case of passing variables to the functions, similar can happen.

> When a variable whose memory is allocated on heap goes out of scope, the value will be cleaned up by `drop` unless the data has been moved such that it is now being owned by another variable.

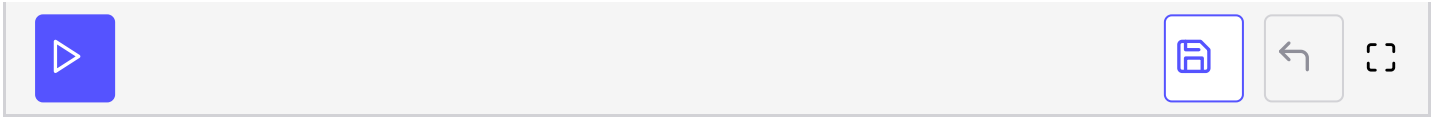
Passing Values to a Function

The ownership of the variable is

- Copied if the value is a primitive data type so the variable can be reused after the function call
- Moved if the value is a non-primitive data type so the value becomes inaccessible after the function call

```
fn main() {
    let str = String::from("Rust"); // str comes into scope
                                    // str is a move type

    pass_string_object(str);        // str's value moves into the function...
                                    // ... and becomes in accessible here
    //println!("{}" , str);         // This line will give an error

    let my_int = 10;                // my_int comes into scope

    pass_integer(my_int);          // my_int value is a copy into the function,
                                    // but i32 is a copy type, so can my used
                                    // use my_int if desired
```

```
} // Here, my_int and then str goes out of scope


fn pass_string_object(my_string: String) { // my_string comes into scope
    println!("{}", my_string);
} // Here, my_string goes out of scope and `drop` frees the memory

fn pass_integer(my_integer: i32) { // my_integer comes into scope
    println!("{}", my_integer);
} // Here, my_integer goes out of scope
```

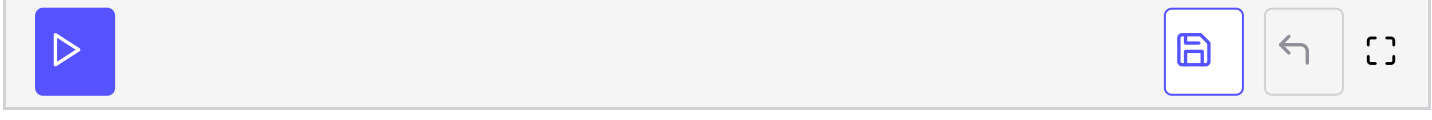


In this example, value `str` of type `String` is moved when passed to the function as an argument and `my_int` of type `i32` is copied.

Return Values from a Function

Returning values from a function transfer the ownership to the caller function.

```
#[allow(dead_code)]
fn main() {

    let str_1 = move_return_value_str_1();  // gives_ownership to str_1

    println!("The function gives ownership to string by returning a value \nstring 1 :{}",str_1);

    let str_2 = String::from("Rust Language");     // assigns a string object to str_2

    println!("This is a string declared \nstring 2 :{}",str_2); // print value of str_2

    let str_3 = moves_str_2_return_str_2(str_2);  // str_2 is moved into the function argument
                                                  // return value moves to str_3
    println!("string 2 passes to the function and returns its value to string 3 \nstring 3 :{}",st

} // Here, str_3,str_2,str_1 goes out of scope respectively
 // str_3 dropped
 // str_2 moved
 // str_1 dropped
fn move_return_value_str_1() -> String {     // gives ownership
                                             // value goes to that calls the function
    let my_string = String::from("Rust"); // my_string comes into scope

    my_string                                // my_string is returned
}

fn moves_str_2_return_str_2(my_string: String) -> String { // my_string comes into
                                                           // scope
    my_string  // my_string is returned
}
```



Here, in this example, variable `str_1` gains the ownership of a String when the value is returned from the function `move_return_value_str_1`. Variable `str_2` is declared and its value is passed to the function `moves_str_2_return_str_2`. Upon being returned from the function the value is saved in `str_3.

> **Note:** `str_2` becomes inaccessible since its value is moved in the function

---

What if you don't want to move the value, and give read-only access. Let's discuss borrowing in the next lesson.