Copy Type and Moved Type

This lesson discusses what happens in tems of memory when variables are assigned to another variable.

We'll cover the following

Copy Type

The ownership state of the original variable (whose value is assigned to another variable) is set to *copied state*. This means that the value of the assignee variable is copied to the assigned variable. A copy of the value is created so that the assignee variable gets the ownership of the value but both the variables have their own copies.

- Variable assignment in case of primitive data type is a copy type.
- pass by value in a function call is an example of copy type

```
let a=1;

let b=a;
```

```
let a=1;

let b=a;
```

2 of 2

> **Why is a primitive type copied?**
>
> Primitive types are stored on the stack and it's fast and cheap to copy them.

Example 1

The following example creates a variable `a` of type `int` and assigns the value of `a` to variable `b`.

> **Note:** The value of `a` is *copied* to `b`

```
fn main() {
    let a = 1;
    let b = a; // copy of 'a' is created
    println!("a:{} , b:{}", a, b); // print 'a' and 'b'
}
```

Example 2

The following example creates a variable `a` of type `array` and assigns the value of `a` to variable `b`.

> **Note:** The value of `a` is *copied* to `b`

```
fn main() {
    let a = [1,2,3];
    let b = a; // copy of 'a' is created

    println!("a:{:?} , b:{:?}", a, b); // print 'a' and 'b'
}
```

Moved Type

The ownership state of the original variable (whose value is assigned to another variable) is set to *moved*.This means that the original variable binding cannot be accessed.

- Variable assignment in case of Non-primitive types such as String Object and Vectors is a moved type.

```
let a=String::from("Rust");
let b=a;
```

1 of 2

```
let a=String::from("Rust");
let b=a;
```

2 of 2

> **Why is non-primitive type moved?**
>
> Non-primitive types are stored on the **heap**. When one variable is assigned to another variable, two variables will point to the same value. This can't happen since it violates ownership rule 2 - if one string/vector will have two owners and one of them changes the value of string there is no way for the other to know. When there is time to clean up the memory, both will try to find the string to clean it. This would lead to memory corruption. To avoid this, the Rust compiler moves the owner to the assigned variables and makes

the other one inaccessible.

Example 1

The following example creates a variable `a` of type `String` and assigns the value of `a` to variable `b`.

> **Note:** The following code gives an error, ✕, since the value of `a` is moved to `b` and `a` becomes inaccessible.

```
fn main() {
    let a = String::from("Rust");
    let b = a; // moves value of 'a' to 'b'
    eprintln!("a:{} , b:{}", a, b); // Error use of moved value 'a'
}
```

Example 2

The following example creates a variable `a` of type `vector` and assign the value of `a` to variable `b`.

> **Note:** The following code gives an error, ✕, if the commented statement is uncommented since the value of `a` is moved to `b` and `a` becomes inaccessible.

```
fn main() {
    let a = vec![2, 4, 8];
    let b = a; // move value of 'a' to 'b'
    println!("b : {:?} ", b); // prints 'b'
    //println!("{:?} {:?}", a, b); // Error; use of moved value: 'a'
}
```

The `clone` Keyword

If you still want both variables to have the same value and be able to use both the variables, it is possible to copy the value of one variable to the other using the `clone` function.

`clone` function.

```
fn main() {
    let mut a = String::from("Rust"); // define a String and save in 'a'
    let b = a.clone(); // b clones a
    a.push('y');
    println!("a:{} , b:{}", a, b);  // print 'a' and 'b'
}
```

Quiz

Test your understanding of copied and moved types in Rust!

Quick Quiz on Copy and Moved Types!

1 Which of the following are copied types?

2 Which of the following are moved types?

Retake Quiz

In the examples above, variables are defined in the `main` function. What happens when the variables are passed as a function argument? How does the ownership concept work? Let's discuss ownership while passing parameters to a function in the next lesson.