1_Ownership_en

May 2, 2023

1 Ownership

All programs have to manage the way they use a computer's memory while running. Some languages have garbage collection that constantly looks for no longer used memory as the program runs; in other languages, the programmer must explicitly allocate and free the memory. Rust uses a third approach: memory is managed through a system of ownership with a set of rules that the compiler checks at compile time.

1.1 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, the value will be dropped.

1.2 Variable scope

[ ]:
```
{                      // s is not valid here, it's not yet declared
    let s = "hello";   // s is valid from this point forward

    // do stuff with s
}
```

String literals are convenient, but they aren't suitable for every situation in which we may want to use text. One reason is that they're immutable. Another is that not every string value can be known when we write our code: for example, what if we want to take user input and store it? For these situations, Rust has a second string type, String. This type is allocated on the heap and as such is able to store an amount of text that is unknown to us at compile time. You can create a String from a string literal using the from function, like so:

[ ]:
```
let s = String::from("hello"); // this type is allocated on the heap
```

1.3 Memory and Allocation

In Rust, 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
```

When a variable goes out of scope, Rust calls a special function for us. This function is called `drop`, and it's where the author of String can put the code to return the memory. Rust calls `drop` automatically at the closing curly bracket.

> **Note:** In C++, this pattern of deallocating resources at the end of an item's lifetime is sometimes called *Resource Acquisition Is Initialization (RAII)*. The drop function in Rust will be familiar to you if you've used RAII patterns.

1.4 Ways Variables and Data Interact: Move

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

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.

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.

If Rust instead copied the heap data as well, the operation s2 = s1 could be very expensive in terms of runtime performance if the data on the heap were large. 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. Freeing memory twice can lead to memory corruption, which can potentially lead to security vulnerabilities.

To ensure memory safety, instead of trying to copy the allocated memory, Rust considers s1 to no longer be valid and, 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:

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 being called a shallow copy, it's known as a move. In this example, we would say that s1 was **moved** into s2.

1.5 Ways Variables and Data Interact: 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);
```

This works just fine and explicitly produces the behavior shown below, where the heap data does get copied.

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 a variable from being valid after we copied it to another variable, like in the following example:

[ ]:
```
let x = 5;
let y = x;
println!("x = {}, y = {}", x, y);
```

Rust has a special annotation called the **Copy** trait that we can place on types like integers that are stored on the stack. If a type has the Copy trait, an older variable is still usable after assignment.

trait = set of methods that must be implemented for a type

Here are some of the types that are *Copy*:

- 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 are also Copy. For example, `(i32, i32)` is Copy, but `(i32, String)` is not.

1.6 Ownership and functions

The semantics for passing a value to a function are similar to those for 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.
main();
```

1.7 Return Values and Scope

Returning values can also transfer 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
}
```

2 Exercises

Exercise 1: Correct the code below so that it works without having to touch the function `into_degrees`.

[ ]:
```
#[derive(Debug)]
struct GeoCoordinate {
    latitude: f64,
    longitude: f64
}
```

```
impl GeoCoordinate {
    fn into_degrees(self) -> Self {
        GeoCoordinate {
            latitude: self.latitude.to_degrees(),
            longitude: self.longitude.to_degrees(),
        }
    }
}

fn main() {
    let radians = GeoCoordinate { latitude: 1.0, longitude: 1.5 };
    let degrees = radians.into_degrees();
    println!("radians = {:?}, degrees = {:?}", radians, degrees);
}

main();
```

[ ]:
```
#[derive(Debug, Copy, Clone)]
struct GeoCoordinate {
    latitude: f64,
    longitude: f64
}

impl GeoCoordinate {
    fn into_degrees(self) -> Self {
        GeoCoordinate {
            latitude: self.latitude.to_degrees(),
            longitude: self.longitude.to_degrees(),
        }
    }
}

fn main() {
    let radians = GeoCoordinate { latitude: 1.0, longitude: 1.5 };
    let degrees = radians.into_degrees();
    println!("radians = {:?}, degrees = {:?}", radians, degrees);
}

main();
```