Rust Course notes

Razafy Rindra, “Hagamena”

July 14, 2022

Contents

1 Introduction

Some quick notes from the Rust book, and the "Let's Get Rusty" online course

2 Common Programming Concepts in Rust

2.1 Variables and Mutability

2.1.1 Mutable variables

Variables in Rust are immmutable by default. In order to createe a mutable variable, we need to add in 'mut' in front of the name like so:

```
    fn main() {
        let mut x = 5;
        println!("The value of x is: {x}");
        x = 6;
        println!("The value of x is: {x}");
}
```

2.1.2 Constants

We can also declare constants in Rust like so:

```
    const my_number: u32 = 100_000;
```

Constants unlike variables can't be mutable, need to be type annotated and they can't be assigned to return value of a function or any value computed at run time.

2.1.3 Shadowing

Shadowing allows us to create a new variable with an existing name, for example:

```
    let x = 5;
    println!("The value of x is: {x}");
    let x = "six";
    println!("The value of x is: {x}");
```

2.2 Data Types

Definition 2.1. Scalar data types represent a single value, **Compound data types** represent a goup of values

Types of scalar data types

1. Integers, they can be 8,16,32,64,128 bit signed or unsigned integers.
2. Floating-point numbers
3. Booleans
4. Character, they represent unicode character

Type of compound data types

1. Tuples; a fixed size array of data that can be of different values.

```
        let tup = ("Let's Get Rusty!",100_000);

```

2. Arrays, in Rust they are fixed length.

2.3 Functions

Functions are defined with an fn keyword like so:

```
    fn my_function(x: i32,y: i32){
        println!("The value of x is: {}", x);
        println!("The value of y is: {}", y);
    }
```

A piece in code in Rust are either a statement or an expression. Statements perform some action but do not return any value, whilst expressions returns values.

```
    fn my_function(x: i32,y: i32) -> i32{
        println!("The value of x is: {}", x);
        println!("The value of y is: {}", y);
        x+y
    }
```

In this function, the println are statements since they don't return anything while 'x+y' is an expression, it is returned by the function (note the last expression of a function is implicitely returned).

2.4 Control Flow

2.4.1 if statements

As in other programming languages:

```
fn main() {
    let number = 3;

    if number < 5 {
        println!("first condition was true");
    } else if number < 22{
        println!("second condition was true.")
    }else{
        println!("condition was false");
    }
}
```

Note, the condition must be a boolean. We can also set a "if-else" statement in a let statemetn.

```
    fn main() {
    let condition = true;
    let number = if condition { 5 } else { 6 };

    println!("The value of number is: {number}");
}
```

2.4.2 Loops

We can create loops with the 'loop' keyword, which will execute the code in it until we call break.

```
    let mut counter = 0;
    let result = loop{
       counter += 1;
       if counter == 10{
           break counter;
       }
    };
```

We can also use the while statement:

```
    let mut number = 3;
    while number != 0{
        println!("{}!", number);
        number -= 1;
    }
```

Finally the third type of loop we can create are 'for loops'

```
    let a = [10,20,30,40,50];

    for element in a.iter{
        println!("The value is: {}", element);
    }
```

We can also loop over a range:

```
    for number in 1..4{
        println!("{}!", number);
    }
```

The last number of the range is excluded.

3 Ownership

What is ownership? The ownership model is a way to manage memory. How do we manage memory today?

1. Garbage collection. Used in high level languages like java or *C#*, the Garbage collection manages memory for you. It's pros is that is error free, you don't have to manage memory yourself so you won't introduce memory bugs. It also faster write time.

   It's cons is that you have no control over memory, it is slower and has unpredictable runtime performance and larger program size since you got to include a garbage collector.

2. Manual memory management

   It's pros are that you have higher control over memory, faster runtime and smaller program size. But it is error prone and has slower writing time.

   Notice that these two have opposite trade-offs.

3. Ownership model

   This is the way Rust manages memory, it's pros are control over memory, faster runtime and smaller program size and is error free (though it does allow for you to opt-out of memory safetey). It's cons is that you have a slower write time, slower than with Manual memeory management, Rust has a strict set of rules around memory management.

Stack and Heap During runetime program has access of stack and heap, stack is fixed size and cannot grow or shrink during runetime and it creates stack-frames for each function it executes. Each stack frames stores the local variables of the function they execute, their size are calculated during compile time and variables in stack frames only live as long as the stack lives.

The heap grows and shrinks during runtime and the data stored can be dynamic in size and we control the lifetime of the data.

Pushing to the stack is faster than allocating on the heap since the heap has to spend time looking for a place to store the data, also accessing data on the stack is faster than accessing data on the heap, since on the heap we must follow the pointer.

```
    let x = "hello";
```

This is a string litteral and has fixed size and is stored in the stack-frame.

```
    let x = String::from(world");
```

x is of type String which can be dynamic in size so it can't be stored on the stack, we ask the heap to allocate memory for it and it passes back a pointer, which is what is stored on the stack.

3.1 Ownership Rules

1. Each value in Rust has a variable called its owner.
2. There can only be one owner at a time.
3. When the owner goes out of scope, the value will be dropped.

Example 3.0.1.

```
        fn main(){
            let x = 5;
            let y = x; // Copy

            let s1 = String::from("hello");
            let s2 = s1; // move (not a shallow copy)

```

```
        println!("{}, world", s1);
    }

```

This returns an error since when we created s2 we made it point to the same "hello" on the heap that s1 points to, but in order to ensure memory safetey, Rust invalidates s1.

What if we do want to clone the string? Use the clone() method:

```
fn main(){
    let x = 5;
    let y = x; // Copy

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

    println!("{}, world", s1);
}
```

3.2 Ownership and functions

Example 3.0.2.

```
    fn main(){
        let s = String::from("hello);
        takes_ownership(s);
        println!("{}", s);
    }

    fn takes_ownership(some_string: String){
        println!("{}", some_string);
    }

```

This gives us an error since after we pass a parameter in a function it is the same as if we assign the parameter to another variable.

So we move s into some_string and after takes_ownership scope is done some_string is dropped.

Example 3.0.3.

```
    fn main() {
        let s1 = gives_ownership();
        println!("s1 = {}", s1);
    }

    fn gives_ownership(){
        let some_string = String::from("hello");
        some_string // returning the string moves ownership to s1
    }

```

Example 3.0.4.

```
    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 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("yours"); // some_string comes into scope

    some_string                              // some_string is returned and
                                             // moves out to the calling
```

```
                                                    // function
    }

    // This function takes a String and returns 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
    }

```

Moving ownership and giving back is tedious, what if we just want to use variable without taking ownership? Use references.

3.3 References

Let us see how references fix the following situation

```
    fn main() {
    let s1 = String::from("hello");

    let (s2, len) = calculate_length(s1);

    println!("The length of '{}' is {}.", s2, len);
}

fn calculate_length(s: String) -> (String, usize) {
    let length = s.len(); // len() returns the length of a String

    (s, length)
}
```

Here in order to calculate the length of the string without taking ownershipe, we need to return a tuple that returns both the string and the length.

```
    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 {
        let length = s.len(); // len() returns the length of a String
        length
    }
```

Here s is a reference of a string and takes no ownership of the string, it points to s1 which points to the string. So when the function finishes executing s is dropped without affecting s1.

Definition 3.1. Passing in references as function paramaters is called **borrowing**. Since we are not taking ownership of the parameters.

Note that references are immutable by default, so how to we modify value without taking ownership?
Mutable references:

```
    fn main() {
    let mut s1 = String::from("hello");

    change(&mut s1);
}

fn change(some_string: &mut String) {
    some_string.push_str(", world");
}
```

Now change can mutate the value of s1 without taking ownership. Mutable reference have a restriction, we can only have one mutable reference to a particular piece of data in a particular scope.

```
    fn main() {
    let mut s = String::from("hello");

    let r1 = &mut s;
    let r2 = &mut s; // Returns an error since can not borrow twice.

    println!("{}, {}", r1, r2);
}
```

This prevents "data races" where two pointers point at the same data and one pointer reads the data and another one tries to write to the data.

What if we try to mix mutable and immutable references? We get another error, we can't have mutable reference if an immutable referece already exists. Immuatble reference don't expect the underlying data to change. But we can have many Immuatble references, since we don't expect the underlying value to change.

Note the scope of a reference starts when it introduced and ends when it's used for the last time, so we can define a mutable reference when all the immuatble references are out of scope(so after we use them for the last time).

Rules of Refences

1. At any given time, we can either have one mutable reference or any number of immuatble reference.
2. References must always be valid.

3.4 Slices

Definition 3.2. Slices let you a contiguous sequence of elements in a collection instead of the whole collection, without taking ownership.

```
    fn first_word(s: &String) -> usize {
    let bytes = s.as_bytes();

    for (i, &item) in bytes.iter().enumerate() {
        if item == b' ' {
            return i;
        }
    }

    s.len()
}

fn main() {}
```

There are two problems with this, the return value of first_word is not tied to the string. If we change the string, the value of the length of the first word doesn't change.

If we wanted to return the second word, we must retrurn a tuple with the index at the start of the word and the index at the end of the word. We have more values we must keep in sync.

Let us use the string slice:

```
    fn main() {
    let mut s = String::from("hello world");

    let hello = &s[..5]; //String Slices, tells us we want the value of the string from index 0 to 4
    let world = &s[6..]; //String Slices, tells us we want the value of the string from index 6 to 10

    let s2 = "hello word"; // String litteral are string slices!

    let word = first_word(s2);

    fn first_word(s: &str) -> &str {
        let bytes = s.as_bytes();

        for (i, &item) in bytes.iter().enumerate() {
            if item == b' ' {
                return &s[0..i];
            }
        }

        &s[..]
    }

}
```

We can also create slice of array

```
fn main(){
    let a = [1,2,3,4,5];
    let slice = &a[0..2]; // This is of type &[i32]
}
```

4 Structs

4.1 Defining and Using Structs

```rust
//Defining a struct
    struct User{
    username: String,
    email: String,
    sign_in_count: u64,
    active: bool,
}

fn main() {
    //Creating an instance of struct
    let mut user1 = User {
        email: String::from("user@email.com"),
        username: String::from("user123"),
        active: true,
        sign_in_count: 1
    };
    let name = user1.username; // Get values from struct with the . operator
    user1.username = String::from("user234"); // modify specific values with . operator

    // Creating a new instance of struct with function
    let user2 = build_user(
        String::from("user345@email.com"),
        String::from("user345")
    );

    //We can also create new instances of a struct using existing instances.

    let user3 = User{
        email: String::from("user456@email.com"),
        username:String::from("user456"),
        ..user2 // Takes the remaining field from user2
    };

    // We can also create structs without naming fields, these are called tuple structs

    struct Colour(i32,i32,i32);
    struct Point(i32,i32,i32);

    // Colour and Point have the same fields, but have different type,
    //so we can't pass a Colour to a function that expects a Point.

}

fn build_user(email: String, username: String) -> User {
    User{
        email, // Since functions arguments have same name as field names, can write it like this...
        username, //...This is called the field init shorthand syntax
        active: true,
        sign_in_count : 1,
    }
}
```

4.2 Methods

```
#[derive(Debug)] // Needed to print out, we will look more into this later.
struct Rectangle{
    width: u32,
    height: u32
}

// Creating implementation block for Rectangle struct,
// This will house function and methodes associated with struct

impl Rectangle{
    fn area(&self) -> u32{ // First arguement in method is always self, the instance the method is being called on
            self.width*self.height
    }

    fn can_hold(&self, other: &Rectangle) ->bool{
        self.width > other.width && self.height>other.height
    }
}

//Struct allow us to have multiple impl blocs

// We can also def associated function, unlike methods they aren't tied to an instance of our struct.

impl Rectangle{
    fn square(size: u32) -> Rectangle{
        Rectangle {
            width: size,
            height: size
        }
    }
```

5 Enums and Pattern matching