Introduction to Rust

1st Semester, A.Y. 2020-2021

The ***Rust programming language*** is a systems programming language focused on three goals: *safety*, *speed* and *concurrency*.

Characteristics[1]

- Rust does not have garbage collection unlike Java.
- It can be embedded in other languages.
- Programs can be written with specific space and time requirements.
- It can be used in writing low level code used for drivers and operating systems.
- It uses compile time safety checks that produce no runtime overhead.

Installation and Compilation

To install rust on your Ubuntu, open the terminal and execute the following command:

```
$ sudo apt-get install curl
$ curl https://sh.rustup.rs -sSf | sh
$ source $HOME/.cargo/env
```

To compile a Rust file, type

```
$ rustc filename.rs
```

To run a Rust program, type

```
$ ./filename
```

Cargo

Cargo is the building and package manager of Rust. It is not required to use, however, it manages building the code, downloading libraries, and building the downloaded libraries, which makes coding in Rust much easier.

To use cargo, you must create a **Cargo.toml** file containing the information similar to the one below:

```
[package]
name = "hello world"
version = "0.0.1"
author = ["dan"]
```

Or, you can ask cargo to create the file for you:

```
cargo new projectname --bin
```

To build and run the files, type

```
cargo build
cargo run
```

1 Basics

For the lab, we will be using Cargo to run our Rust programs. Below is a sample Hello World program in Rust.

```
fn main(){
    println!("Hello World"); //printing!
}
```

A rust program must contain a `main` function just like the C programming language. The main program, containing the `main` function, must be saved as **main.rs** and must be placed in the **src** folder.

[1] Rust programmers are called Rustaceans. Isn't that fun?

A. Comments

There are three ways to put comments in Rust.

1. `//` - used for line comments
2. `///` - used for doc comments
3. `//!` - used for doc in crates, modules and functions

Line comments are the same as normal comments in other languages. Doc comments, on the other hand, are used, together with `assert` functions, to later on generate documentation files and/or manuals for the program. These documentation files can be generated using the `rustdoc` command.

B. Variables

Variables in Rust are declared using the `let` keyword. Rust, by default, creates **immutable** (cannot be modified) variables, however, you can bypass this using the `mut` keyword.

```
let x = 5;       //immutable
let mut y = 10; //mutable

println!("x:{}, y:{}",x,y); //will print x:5, y:10
//x=10; //this will produce an error
y=15; //this is valid

//the print below will work if we comment x=10;
println!("x:{}, y:{}",x,y); //will now print x:5, y:15
```

Even though you declare a variable as immutable, you can still change its value by **shadowing**:

```
let x = 5;
println!("x:{}",x); //will print x:5

let x = x+5;         //this x 'shadows' the previous x
println!("x:{}",x); //will print x:10

let x = "five";      //shadowing also allows you to change the type of the var
println!("x:{}",x); //will print x:five
```

2 Data Types

A. Scalar Data Types

Rust has a wide array of data types.

A..1 Boolean

The built-in boolean type, `bool`, having two values: `true` and `false`.

```
let x:bool = true;
let y = false;
```

These values are used to evaluate conditional statements.

A..2 Character

Represents a single character. Unlike most languages where a character is only 1 byte, Rust uses 4 bytes for a character which allows you to put non-ascii characters into your code.

```
let letter = 'a';
let symbol = '%';
let omega = '\u{03A9}'; //unicode for omega symbol
```

A..3 Numeric Types

Rust numeric types consist of two parts: the category and the size. The category can be signed or unsigned, fixed or variable, and floating-point or integer. For the size, the more bits used, the larger the number it can represent.

Below is the list of the different numeric types:

- `i8` - 8-bit signed integer (fixed)
- `i16` - 16-bit signed integer (fixed)
- `i32` - 32-bit signed integer (fixed)
- `i64` - 64-bit signed integer (fixed)
- `u8` - 8-bit unsigned integer (fixed)
- `u16` - 16-bit unsigned integer (fixed)
- `u32` - 32-bit unsigned integer (fixed)
- `u64` - 64-bit unsigned integer (fixed)
- `isize` - variable-size signed type
- `usize` - variable-size unsigned type
- `f32` - 32-bit floating-point (fixed)
- `f64` - 64-bit floating-point (fixed)

```
let x = 12;  // default is i32
let y = 5.8; // default is f64
let z:u8 = 255; // unsigned 8-bit integer range 0-255, valid

//let a:i8 = 255; // out of range for i8 (-128 to +127), produce an error
println!("{}", std::i8::MAX); //check the max allowed value for i8 (+127)
println!("{}", std::i8::MIN); //check the min allowed value for i8 (-128)

let mut b:i32 = 64;
b = 478; //valid since b is mutable
```

B. Compound Data Types

Compound types can group multiple values into one type.

B..1 Tuples

A tuple is a general way of grouping together some number of other values with a variety of types into one compound type. They have a fixed-length all throughout program execution.

They are created using parenthesis, and each element can be accessed using the dot (`.`) operator and by following zero-indexing.

```
let tup = ('a',32,true);
println!("Second element: {}", tup.1); //will print 32
println!("The tuple contains: {:?}", tup); //will print ('a', 32, true)

let (x,y,z) = tup; //destructuring
println!("The tuple contains: {}, {}, {}",x,y,z); //will print a, 32, true
```

B..2 Arrays

Arrays represent a fixed-size list of elements of the **same type**. By default, arrays are immutable.

```
let arr1 = [1,2,3,4];
// arr1[0] = 3; will result to an error because arr1 is immutable

let mut arr2 = [2,3,4];
arr2[0] = 1; // will not produce an error because arr2 is mutable
println!("{:?}", arr2); //will print [1, 3, 4]

let a = ['g';20]; //creates 20 elements all having a value of zero
println!("Size of a is: {}.",a.len()); //will print 20
println!("{:?}", a); //will print ['g', 'g', 'g', 'g', 'g', ... and so on]
```

B..3 Slices

Slices are references to another data structure. They are useful for allowing access to a portion of an array without copying the portion. We'll discuss references with more detail later at Section 8.C. For now, creating slices would need the ampersand (&) operator and the range of the slice.

```
let arr1 = [1,2,3,4,5];

let all = &arr1[..]; //will contain everything
let some = &arr1[1..4]; //will contain 2,3,4.
// range of slice covers indexes [1,4).
// this can also be used in Strings and Vectors
```

C. Collections

Collections are built-in data structures in Rust. Unlike Arrays and Tuples, which are stored in the stack and whose size must be known at compile time, collections are stored in the heap and can grow or shrink as the program runs.

C..1 Vectors

Vectors are like arrays but can grow or shrink during execution. Below is a sample usage for vectors in Rust:

```
let v:Vec<i32> = Vec::new(); //creates an empty vector of type i32
let v = vec![1,2,3]; //creates a vector containing 1,2,3 of type i32

let mut v1 = Vec::new();
v1.push(5);  //pushes 5 in the vector
v1.push(4);  //pushes 4 in the vector
v1.push(3);  //pushes 3 in the vector
println!("{:?}", v1); //prints [5, 4, 3]
println!("v1[0]: {}", v1[0]); //prints v1[0]
println!("v1[0]: {}", &v1[0]); //uses slicing
println!("v1[0-1]: {:?}", &v1[0..2]); //uses slicing; prints [5, 4]
//println!("v1[0]: {}", v1.get(0)); //ERROR
              // because get returns Some(value) or None, {} can't handle it

//HOWEVER, if the index does not exist...
//println!("v1[10]: {}", v1[10]); //this will panic
//println!("v1[10]: {}", &v1[10]); //this will panic
println!("v1[10]: {:?}", v1.get(0)); //this will NOT produce an error
             // assuming you use {:?}

v1.pop(); //removes the last value added
```

C..2 Strings

Strings in Rust are implemented as collection of bytes (that's why strings are under the collection types). Most of the operations available for vectors are available for strings.

Below are examples of string usage in Rust:

```
let mut s1 = String::from("hello");
let mut s2 = "hello".to_string();
let mut s3 = String::new(); //new empty string
s3 = "str".to_string();     //puts a string in the new string
s3.push_str("ing");         //appends ing to s3 (appends the string)
s3.push('s');               //appends one letter to s3 (appends a character)
```

C..3 Hash Maps

Hash maps stores a mapping of keys to values. For the lab, we will not be discussing hash maps.

3 Operators

Operator	Description
=	Assignment
Arithmetic	
+	Addition
-	Subtraction / Negation
*	Multiplication
/	Division
%	Remainder
Bitwise	
!	Complement
&	Bitwise AND
^	Bitwise XOR
|	Bitwise OR
<<	Left-shift
>>	Right-shift
Logical	
!	Negation
&&	Logical AND
||	Logical OR
Comparison	
!=	Non-equality
<	Less than
>	Greater than
<=	Less than or equal
>=	Greater than or equal

4 Input and Output

As previously shown, printing in Rust uses the `print!()` or `println!()` functions. The only difference is that `println!()` adds a newline.

```
println!("Jane Doe"); //printing simple strings
println!("Jane {}", "Doe"); //printing single data
println!("{:?}", ["Jane", "Doe"]); //printing data from arrays, tuples, etc.
```

For getting input, one must first include the library for input.

```
use std::io;  // library for input (standard input output)

fn main(){
     let mut name = String::new();
     println!("Enter name: ");   //printing
     io::stdin().read_line(&mut name).expect("Error");  //wait for input
     println!("My name is {}",name);
}
```

Inputs in rust are returned as strings. They must be explicitly typecasted to their appropriate data type before using with other operations. This is shown below:

```
use std::io;

fn main(){
   let mut num = String::new();
   io::stdin().read_line(&mut num); //waiting for input
   //creating a new var where num is trimmed and converted based on what
   //is parsed and what is the type specified
   let num:isize = num.trim().parse().expect("error");
   println!("{}",num);
}
```

5 Control-Flow Statements

A. Conditional Statements

The syntax of `if-else` statements in Rust is almost the same in C, except in Rust, you can drop the parenthesis in the condition statements. Below are some examples:

```
let grade = 55; //or input from user
let lecabsences = 6;
let lababsences = 4;

if (lecabsences > 7) || (lababsences > 3){
   println!("Grade is 5.0");
}else if grade>=55{
   println!("You passed CMSC 124!");
}else{
   println!("Grade is 5.0");
}

//if-else can also be used in the let statement
let condition = true;
let num = if condition { 5 } else { 6 }; //valid
//the values inside each block must have the SAME TYPE
```

NOTE: Conditional statements must result to a `bool` value. If the `if` statement was succeeded by an non-`bool` value, an error will be produced.

B. Iterative Statements

Repeatition of statements can be done in three ways:

1. **Loop**
   This repeats the statements inside the loop block and only terminates if it encounters a termination condition.

   ```
   let mut i = 0;
   loop {        //repeats whatever is in the curly braces
       i += 1;
       println!("{}", i);
       if i==10 {
         break; //will loop infinitely if the break is not encountered
       }
   }
   ```

2. **While**
   This repeats the statements inside the while block and only terminates if the condition evaluates to `false`.

   ```
   let mut i = 0;
   while i != 10 {  //condition must evaluate to bool
      i+=1;
      println!("{}",i);
   }
   ```

3. **For**
   The `for` loop can be used to iterate over a collection or a range. The `break` and `continue` keywords can also be used in `for` loops.

   ```
   for i in 1..11 {  //range (start to end-1)
      println!("{}", i);
   }

   let v1 = vec![1,2,3,4,5];
   //iterating over v1, however, since v1 is not mutable
   //we cannot change the value of its elements
   ```

```
for i in &v1 {
   print!("{} ",i);
}

//iterating over a mutable vector and changing its contents
let mut v2 = vec![1,2,3,4,5];
for i in &mut v2{ // & means i is just a reference to the original data
   *i = *i + 10; // * accesses to values of a reference (like pointer)
}
println!("{:?}", v2); //will print the new values
```

6 Functions

Functions in Rust are declared using the `fn` keyword. Rust follows *snake case*[2] for function names and variable names. Below is the format for function declaration and function calls:

```
fn function_name(variable_name:data_type) -> return_value_type {
     code_block
}
//function call
function_name(actual_parameter);
```

Below are sample declarations:

```
fn print_sum(x:i32, y:i32) {
   println!("{} + {} = {}",x,y,x+y); //will print 5 + 6 = 11
}

fn return_sum(x:i32, y:i32) -> i32 {
   x+y  //DO NOT put a semicolon on the return value!
}

fn main(){
   print_sum(5,6);
   let x:i32 = return_sum(5,6);
   println!("{}",x); //will print 11
}
```

7 Structures

Struct, or structure, is a custom data type that lets you name and package together multiple related values. This is similar to the `struct` construct in C. Below is the format for structure declarations:

```
//used to create basic structures
struct structure_name {
  variable_name: data_type,
  variable_name: data_type,
  variable_name: data_type
}

//used to create tuple structures; do not need a name for its fields
struct structure_name(data_type, data_type, data_type);
```

Below are example structs in Rust:

```
struct Color1 { //Color1 is the name given by the programmer
  red: i32,     //Color has three(3) fields
  green: i32,
  blue: i32
}
```

[2] All letters are lowercase and words are separated by underscores

```
struct Color2(i32, i32, i32); //Tuple structure, no field name needed
```

Structure fields in Rust can be accessed the same way as structures in C—using the dot (.) operator.

```
fn main() {
  let cerulean = Color1{
    red: 42,
    green: 82,
    blue: 190
  };
  println!("RGB: ({}, {}, {})", cerulean.red, cerulean.green, cerulean.blue);

  let maroon = Color2(128, 0, 0);
  println!("RGB: ({}, {}, {})", maroon.0, maroon.1, maroon.2);
}
```

Here are some additional examples to structures:

```
struct User {  //User is the name given by the programmer
  username: String,     //User has four(4) fields
  email: String,
  sign_in_count: u64,
  active: bool
}

fn create_user(email:String,username:String) -> User{
   User{
      email: email, username: username, active: true, sign_in_count: 1
   }
}

fn main(){
   //create a user using the User structure
   let mut User1 = User{
      email: String::from("jmdelacruz@up.edu.ph"),
      username: String::from("jmdelacruz"),
      active: true,
      sign_in_count: 1
   };

   //change the value of the sign_in_count field
   User1.sign_in_count = 2;

   //create a user using the function
   let User2 =
      create_user("absy@up.edu.ph".to_string(),String::from("absy"));

  //this doesn't work for structures, you must access the data individually
  println!("{:?}", User1);
}
```

8 Ownership System

Ownership is the central feature of Rust. It allows the programmer to safely manage memory without making the program speed slow.

Rust has three rules in the ownership system:

1. Each value in Rust has a variable that's 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.

A. Variable Scoping

A variable's scope is the lines of program where it is allowed to be accessed. Usually, the scope of the variable starts when it is declared and ends when its owner terminates.

```
fn sum(x:i32,y:i32) -> i32 { //x and y's scope starts here
  x+y
} //x and y's scope ends here
//greeting's scope does not cover the sum function

fn main(){

  if sum(5,6) > 10{
    let greeting = "hello"; //scope of greeting starts
  }             //scope of greeting ends

  //println!("{}",greeting); // IS THIS VALID?
  // NO! Because greeting went out of scope after the curly brace
  // of the if statement
}
```

B. Moving

For basic data types live variables, assignment of a variable to another variable will create a copy of the value. However, this is not the case for other data structures. Take the example below:

```
let x=12;
let y=x;  //This is valid since y will be given a copy of x
println!("{} {}",x,y); //will print 12 12

//but let's say we have a string
let s1 = String::from("hello");
let s2 = s1; //this is still valid
println!("{}",s2); //will print hello, still valid
println!("{}",s1); //INVALID!
```

Why is the last line invalid even though `let s2=s1;` was allowed?

Remember rule #2 of the ownership system. When `let s2=s1;` was executed, we *moved* the ownership of the value of `s1` to `s2`. Thus, we cannot use `s1` anymore because it does not own anything.

Why does Rust do this? Remember rule #3—once a variable goes out of scope, its value is freed. If `s1` goes out of scope, then the value of `s1` will be freed (thus, "hello" will be freed). Then, if `s2` also goes out of scope, it should free its value. BUT! "hello" has already been freed before when `s1` went out of scope. Thus, having multiple owners for the same values will create **double-free**[3] errors.

This moving scheme also applies to parameters passed unto functions. If we want to get the ownership back from the function, we can return the passed parameter (if multiple return values, one may use tuple destructuring), or use references. Let's see an example:

```
fn return_ownership(moved_string: String) -> String {
  moved_string //the string is moved to the variable to catch the return
}

let s1 = String::from("hello");
let s2 = return_ownership(s1); //s1's value is moved to the parameter
```

[3] Is your CMSC 21 ~~nightmares~~ memories being relieved?

C. References and Borrowing

References can be created using the ampersand (`&`) operator. References allow you to refer to some value without taking ownership of it. The opposite of referencing is called *dereferencing* which uses the asterisk (`*`) operator.[4]

```
fn get_length(s: &String) -> usize { //has an & to signify that
    s.len()                          //it is borrowed
}
fn main(){
    let s1 = String::from("hello");
    let x = get_length(&s1); //get_length has no right to free the value of s1
    println!("Length of {} is {}.",s1,x); //valid because s1 is only borrowed
}
```

What if we want to change what we borrowed? We just need to pass a mutable data and specify that it is mutable.

```
fn append_world(s: &mut String) { //add mut here
    s.push_str(" world")
}
fn main(){
    let mut s1 = String::from("hello"); //declare the variable as mutable
    append_world(&mut s1);          //add mut here
    println!("New string: {}.",s1); //and it is mutated!
}
```

HOWEVER! Only one variable is allowed to borrow another **mutable** variable per scope. If you want multiple references, you are allowed to as long as they are all immutable.

```
let mut s = String::from("hi");
{ //scope 1
let s1 = &mut s; //valid, s1 is currently the only borrower
let s2 = &mut s; //invalid! s is already borrowed as mut in this scope
}

{ //scope 2; VALID because the borrowing is not mutable
let s3 = &s; //s3 and s4 is not allowed to change s, so this is fine
let s4 = &s;
}
```

This restriction allows Rust to make sure that there will be no *data race condition*[5] in the program, since only one reference can be used to change the data.

The last thing you need to remember about references are **dangling references**. Dangling references are pointers that points to memory locations that may have been given to another process already. In Rust, the compiler guarantees that your references will never be dangling references.

```
fn get_pointer()-> &String { //returns a reference to a String
    let s = String::from("hi");
    &s    //once this is returned, s will be dropped because
}         //it'll be out of scope

fn main(){
    let ptr = get_pointer();
    let p; //ANOTHER EXAMPLE OF DANGLING REFERENCE
    {
        let num = 10;
        p = &num;
    }
    println!("{}",p); //this is invalid because the value being pointed by p
                      //is already freed when we went out of the scope of num
}
```

[4] Isn't this CMSC21 all over again? Yes. But more secure.
[5] A **data race condition** is a condition where two variables might change the value of a variable at the same time.

D. Lifetimes

The last thing you need to consider in Rust's ownership system is *lifetimes*. Lifetime is the duration where a certain reference is valid or *alive*.

Lifetimes main use in Rust is to solve the dangling pointer problem. Consider the example below:

```
//v1 and v2 is borrowed; will return what is borrowed
fn get_bigger(v1: &Vec<isize>, v2: &Vec<isize>) -> &Vec<isize>{
    if v1.len() < v2.len() {
        v2
    }else{
        v1
    }
}

fn main(){
    let v1 = vec![1,2,3];
    let v2 = vec![2,3,4,5,6,7];
    let ptr = get_bigger(&v1,&v2); //let get_bigger borrow v1 and v2
                                   //because we'll use it below
    println!("Bigger of {:?} and {:?} is {:?}.",v1,v2,ptr);
}
```

This will produce an error because Rust doesn't know if the reference being returned is of v1 or v2. Rust needs to know this to check if the lifetime of v1 or v2, if returned, is still valid. But we also do not know which will be returned because of the `if-else`. Thus, we must give a generic lifetime to the function signature.

Generic lifetime annotation must follow the syntax `'lifetime` and must be placed in the data type and after an ampersand. Below are examples:

```
&i32  //a reference to an i32 value
&'a i32 //a reference to an i32 value with lifetime 'a
&'a mut i32 //a reference to an mutable i32 value with lifetime 'a
```

To correct our previous example, we must put lifetime annotations in the function signature as shown below:

```
//put 'a in all formal parameters and return data types
//put also the <'a> after the function name to tell Rust
//   that the function, for some lifetime 'a, takes two parameters
//   that also lives during lifetime 'a.
fn get_bigger<'a>(v1: &'a Vec<isize>, v2: &'a Vec<isize>) -> &'a Vec<isize>{
    if v1.len() < v2.len() {
        v2
    }else{
        v1
    }
}
```

References

[1] Getting Started with Rust. https://doc.rust-lang.org/book/second-edition/ch01-00-getting-started.html.

[2] The Rust Programming Language, 2018. https://doc.rust-lang.org/book/2018-edition/.

[3] Tan, K. Introduction to Rust. Previous CMSC 124 Handout.

[4] Traversy Media. Rust Crash Course — Rustlang . [Youtube video]. https://www.youtube.com/watch?v=zF34dRivLOw.

[5] YouCodeThings. Learning Rust: Memory, Ownership and Borrowing. [Youtube video]. https://www.youtube.com/watch?v=8M0QfLUDaaA.