MKProjects 2021

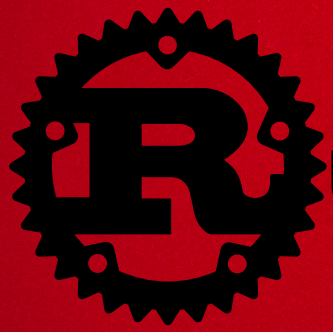
RUST BASICS

A Developer's Favourite Language

MUSTAFIF KHAN

PUBLISHED BY MKPROJECTS

MKPROJ.COM



Rust Basics

Contents

Rust Basics

1. Getting Started

Rust is a systems programming language that combines the speeds of a low level programming language, as well as the memory safety with a high programming language. This makes Rust a very impressive language as these aspects are an either or, however Rust is able to reach this with it's ideals of Ownership, and the fact that it does not use a garbage collector.

Rust was developed by Graydon Hoare at Mozilla Research, and it first publically appeared at July 7, 2010, since then it developed its own foundation funded by many corporations, and is called the Rust Foundation, .It has gained remarks of a language not only similar in syntax to C++, but could possibly start replacing it in a company's application for more safe and secure code. This has proved to be true for MKProjects as the first command line project was written in C++, but however the reliability of an infinite loop is anything but reliable, so looking into Rust has been a good decision as it now runs most of the projects, even this book is created using Rust!!!

Install

To install Rust, it is recommended to install via Rustup so visit rustup.rs

Rust Compiler

In the first few sections we will be utilizing the Rust Compiler, and it used as `rustc file.rs` which will create a binary that can be executed. In this book, it will reflect upon the linux terminal, however Unix Systems like MacOS will not differ, and Windows will have `.exe` files instead of binary files.

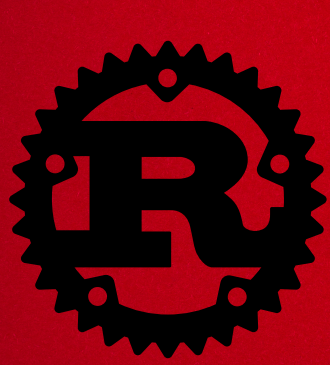
Extra Resources

Keep in mind this is only an entry point into Rust and obviously doesn't cover everything that Rust offers, (and trust me there's a lot), so it is good to look into extra resources, and good for you that the Rust Foundation already has you covered.

- The Book
  - `https://doc.rust-lang.org/book/`
  - This is a good introduction to Rust and can cover more topics in better detail
- Rust By Example

    - `https://doc.rust-lang.org/stable/rust-by-example/`
    - Learn Rust in a more interactive way with examples
- Command Line Book
    - `https://rust-cli.github.io/book/index.html`
    - Create Command Line applications using Structopts
- WebAssembly Book
    - `https://rustwasm.github.io/docs/book/`
    - Create Web applications with Rust and Webassembly
- Embedded Book
    - `https://doc.rust-lang.org/embedded-book`
    - Created programs for embedded system with Rust

There is many more books provided by the Rust Foundations, so make sure to check out it out at `https://www.rust-lang.org/learn`



ust Basics

2. Bindings & Mutability

In Rust, another way of calling a binding is a variable. A binding is used as a placeholder to store memory of a particular value, and in Rust, by default is set to immutable. Before talking more about this, let's first look at how to declare a binding, one with type inference, and the other with an explicit type:

```
let a = 5.0; //Inferred by default as float64 or f64
let b: f64 = 5.0; //Explicitly declared as type f64
```

To delcare a binding, you must use the `let` keyword with a purposeful name and an assignment operator, =, with an appropriate value with it. Due to Rust being a safe memory language, all bindings are immutable by default, and this is different than most languages, however you'll see most of the times mutability isn't necessary. But that doesn't mean mutability isn't possible, it is instead initiated with the convenient `mut` keyword.

```
fn main(){
    let mut a = "I am Mutable";
    a = "I can change values!!!";

    let b = "I am immutable";
    b = "This will cause an error!";
    //Press play to check the error!
}
```

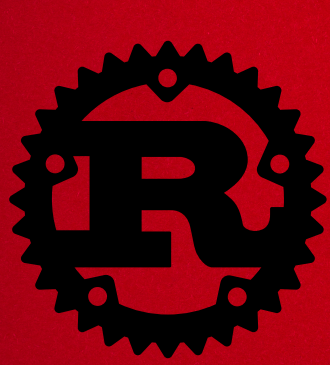
2.1 Data Types

Rust has four main data types, which are the following:

- Integers (non decimal point numbers)
  - Signed: `i8, i16, i32, i64, i128, isize`
  - Unsigned: `u8, u16, u32, u64, u128, usize`
- Floating Points (decimal point numbers)
  - `f32 or f64(inferred default)`
- Boolean

  - `bool`, either evaluates to `true or false`
- Characters
  - `char`, such as `'L'`

Another type that is commonly used but not primitive is Strings and is denoted by `String`.



ust Basics

3. Imports & Namespaces

In many languages it is common to import libraries from external, or standard libraries, in our case we will only consider the standard library until we talk about [Cargo in Section 6](06-cargo.md). So I guess the best way to start is by first figuring out how to import a library, and that is done by the `use` keyword.

To help with this section, we will create a simple user input program by using the standard library input/output module ,!std::io!.

```
use std::io;

fn main() {
    let mut input = String::new(); // Creates a new String
    io::stdin().read_line(&mut input).unwrap();
    println!("Your input is {}", input);
    // What this means is that we borrow (&) the binding input, and
    // it must be mutable, since it will change it's value

}
```

In this example, we specify that we only want to use the `io` module in the `std` library. However let's say we don't know what we really want from the `std` library, and instead of guessing, we could just import everything by using `*` after `std::`.

Consider the example below:

```
use std::*;
fn main() {
    //We will be copying one file to another
    let mut a = fs::File::create("file.txt")
        .expect("file not created");
    //The file we will be copying from
    let mut b = fs::File::open("copy.txt")
        .expect("file not found");
    //Using io to copy b to a
    io::copy&mut b, &mut a)
        .expect("file cannot be copied");
}
```

We get the following result:

```
$ rustc ex.rs
$ ./ex

#copy.txt
This file has words!!!

#file.txt
This file has words!!!
```

As you can see we could access anything from `std` using `*`, however it isn't the most effective thing to do, in this example we only needed `io` and `fs` modules, so let's just import those.

```
use std::{fs, io};

fn main() {
    //We will be copying one file to another
    let mut a = fs::File::create("file.txt")
        .expect("file not created");

    //The file we will be copying from
    let mut b = fs::File::open("copy.txt")
        .expect("file not found");
    //Using io to copy b to a
    io::copy(&mut b, &mut a).expect("file cannot be copied");
}
```

As you can see, if you know what you want to import, and need multiple modules, then surround them with `{ }`.

When it comes to namespaces, whatever the module you import is the name you must start with. Such as, `std::io` you use `io::`, but if you did `std::fs::File`, you must start with `File::`

4. Control Flow

4.1 If Statements

An `if` statement is a block of code that can be executed by a program if a condition results to `true`. For example let's ask the user for a number from 1 to 10, if it's less than 5, we will say `"Wow that's low huh"`!.

```
use std::io::stdin; //To get standard input

fn main(){
    let mut input = String::new();
    println!("Enter a number from 1 to 10!");
    stdin().read_line(&mut input).unwrap(); //Gets user input

    //Now to convert the String to i32
    let i:i32 = input.trim().parse().expect("Expected an integer");

    if i < 5 {
        println!("Wow that's low huh!");
    }
}
```

As you can see, we have the `if` keyword followed by the condition, such as `i < 5` then the block of code executed is surrounded by `{ }`.

4.2 Else Clause

This program isn't really that good, it only gives you something if you guess a number less than 5, but let's say I guess 8? I should still get something, and that's when the `else` clause comes in, unlike the `if` statement, it will execute a block of code only if it results to `false`. So let's fix our program so when I guess 5 or greater I am presented with `"Buddy you guessed too high"`!.

```rust
use std::io::stdin; //To get standard input

fn main(){
    let mut input = String::new();
    println!("Enter a number from 1 to 10!");
    stdin().read_line(&mut input).unwrap(); //Gets user input

    //Now to convert the String to i32
    let i:i32 = input.trim().parse().expect("Expected an integer");

    if i < 5 {
        println!("Wow that's low huh!");
    } else {
        println!("Buddy you guessed too high!");
    }
}
```

Notice how when you're using the `else` clause, you don't require a condition, since it's like a default.

4.3 Else if Statement

Now I got another problem with my program, I want something to happen when they guess 5, so how do I do that, add another `if` statement? But that would look so disgusting, so instead why not add an `else if` statement? An `else if` statement is like a second order condition, if the first `if` statement results to false, then the `else if` statement is checked, and if it results to `true` then it's block of code is executed.

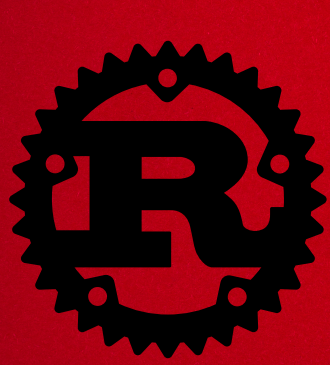
```rust
use std::io::stdin; //To get standard input

fn main(){
    let mut input = String::new();
    println!("Enter a number from 1 to 10!");
    stdin().read_line(&mut input).unwrap(); //Gets user input

    //Now to convert the String to i32
    let i:i32 = input.trim().parse().expect("Expected an integer");

    if i < 5 {
        println!("Wow that's low huh!");
    } else if i == 5{
        println!("You guessed right!!!");
    } else {
        println!("Buddy you guessed too high!");
    }
}
```

An `else if` statement has the same syntax as an `if` statement, except it must be after an `if` statement and before the `else` clause.



ust Basics

5. Modules & Functions

5.1 Function Syntax

A function in Rust is defined with the `fn` keyword and is a block of code that uses parameters to pass arguments and returns a value or values. To create a function first use the `fn` keyword along with a name, then put brackets!()! that will contain the parameters, and then `{}` for your block of code.

To make this a lot simpler, observe a simple `hello_world()` function that asks for a name and prints `"Hello World Name"`!. To do this we will also require a parameter name that's a `String`.

```
//Let's define our hello_world function
fn hello_world(name: &String){
// Since we don't want to entirely use the variable
// We will borrow it using &
    println!("Hello World {}!", name);
}

fn main(){
    let mut name: String = String::from("Bob");
    //To call a function simply use it's name
    hello_world(&name);
}
```

5.1.1 Return Type

When you write a function in Rust, you can either return nothing, in other languages that may be called a `void` function, or you can return a data type using !fn foo()->type!. For our example we will create a simple add function to show an easy example to return:

```
fn add(a:i32, b:i32)->i32{
    a + b
}
//Another way of returning is:
fn add_alt(a:i32, b:i32)->i32{
    a + b
```

```
}
fn main(){
    let a = add(2,5);
    let b = add_alt(2,5);

    if a == b{
        println!("Same result");
    }
}
```

5.2 Modules

To introduce modules we will be writing a program that involves two files, we will respectively have `lib.rs` and `main.rs`. We will have two different sections or modules in our library, one being for geometry of 2D shapes and the other for 3D. To define a module, we must use the `mod` keyword.

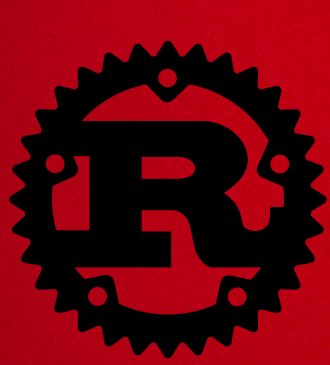
To have functions able to use outside of the module or library, make sure to use the `pub` keyword to make it public (private by default).

```
//lib.rs
mod 2D{
    pub fn Area(len: f64, width: f64)->f64{
        len*width;
    }
}
mod 3D{
    pub fn Volume(len: f64, width: f64, height: f64){
        len * width * height;
    }
}
```

Now to import our library, we must first `mod` it then we can `use` the library, so let's do that.

```
//main.rs
mod lib;
use lib::2D;
use lib::3D;

fn main(){
    /*
    Let's find the area and volume of
    a square and cube respectively.
    */
    let side = 2.56;
    println!("Area: {}", Area(side, side));
    println!("Volume: {}", Volume(side, side, side));
}
```



ust Basics

6. Cargo

Cargo is Rust's package manager utility and is the recommended way to build a Rust program as it allows the ability to import crates, and publish your own. It is very convenient to the user, and offers two different type of projects to be built:

- bin : A binary project is a project to be executable (default)
- lib: A library is a way to store all your functions, modules, etc. seperately

A crate in Rust is a cargo project published in Rust's `http://crates.io`. To create a new cargo project, we use the `cargo new` option, and provided a name will create a project for us with the following structure:

```
$ cargo new myproject # Create binary crate myproejct
$ cd myproject
$ ls
Cargo.toml src
```

The `src` directory contains all of our source code, and inside the folder contains a `main.rs` file for us to do our work. The `Cargo.toml` file contains all of the metadata for out project such as name, author, version number, dependencies, etc. For more information about what can be done in Cargo, you can visit `https://doc.rust-lang.org/cargo/index.html`.

For our example, we will make use of the `rand` crate to create randomized numbers for a guessing game.

```
$ cargo new guess_game # create a new guess_game
$ cargo search rand # let's find the version number of the crate
rand = "0.8.4"      # Random number generators and other randomness
    functionality.
```

Let's first edit our `Cargo.toml` file and add the `rand` crate as a dependency.

```
#Cargo.toml
[package]
name = "myproject"
version = "0.1.0"
edition = "2018"

```

```
# See more keys and their definitions at https://doc.rust-lang.org/cargo/
    reference/manifest.html

[dependencies]
rand = "0.8.4"
```

Now that we have rand as a dependency,we are free to use it in our project. Now we can edit our `src/main.rs` file and begin our guessing game that does the following:

- The user only has one attempt to try
- We will tell them if they are too low or too high
- They will have to guess from 1 to 10

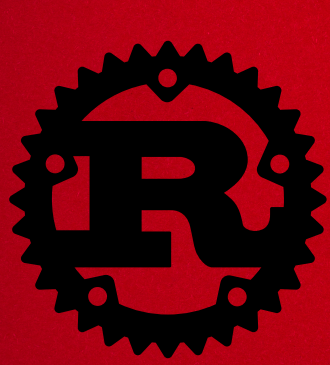
```
use rand::Rng;// For rand numbers
use std::cmp::{Ord, Ordering};// For Ordering
fn main() {
    let rng = rand::thread_rng().gen_range(1..10);//Our random number
    generator

    let mut s = String::new();//Use to get user input
    println!("Enter your guess: ");// Ask for input
    std::io::stdin().read_line(&mut s).unwrap();

    let guess: i32 = s.trim().parse().unwrap();

    //Now we match our guess to compare with our random number
    match guess.cmp(&rng) {
        Ordering::Greater => println!("Too high!"),
        Ordering::Less => println!("Too low!"),
        Ordering::Equal => println!("You're right!"),
    }
}
```

You can try yourself running `cargo run` and see if you guess right. You may also build the project by running, `cargo build`, and the executable will be found in `target/debug/<name>`.



Rust Basics

7. Ownership Basics

Rust follows a memory/thread safe, zero abstraction model and due to that a Garbage Collector won't comply. A GC automatically handles the drops and move between data in the program, and due to that it costs some performance for that memeory management.

So what do we do then? Well Rust uses ownerships rules to comply with Rust's Borrow Checker that ensures memory safety, however in this book we won't go into much detail of Rust's handling with stack and heap allocation, but we will show enough to avoid fighting the borrow checker.

7.1 Ownership Rules

- Each value in Rust has a variable that's called it's *owner*
- There can only be **one** owner at a time
- When the owner goes out of scope, the value will be **dropped**

We can see a simple example using closures:

```
{ //my_var isn't valid, isn't declared yet'
    let my_var = "My Variable exists here!"; //valid from this point
    forward
    //Do stuff with my_var
} //The scope is over, so my_var is dropped
```

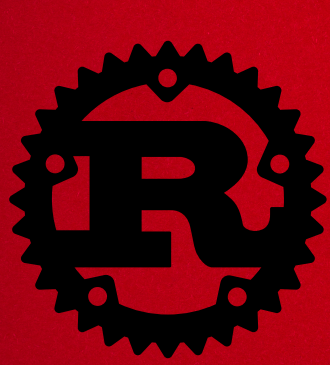
Now we introduce the concept of borrowing, and this is done by using the `&` operator. Borrowing allows other variables to use a variables data, and we have two ways of borrowing, we have `&` (immutable borrow) and `&mut` (mutable borrow) where mutable borrow allows for us to manipulate the data.

```
// This program borrows a vec and pushes 2 into it
fn push_two(v: &mut Vec<i32>){v.push(2);}
fn main(){
    let mut v = vec![1,6,7,8];
    push_two(&mut v);
    println!("{:?}", v)
}
//Result: [1, 6, 7, 8, 2]
```

7.2 Strings & Strs

If you havent noticed, but Rust has two different type of string types, `String` and `&str`, and this can be explained due to ownership. A `String` variable is an owned value, and this means if you move the value from one variable to another, the new variable now owns the value and the other variable is dropped. Now the other string type `&str` is the reference to a string, or a borrowed version. So if you move one value to the other, both are still valid, and these both have it's use cases and it's important to note that in functions it is common to use `&str` instead of `String` (due to borrowing the parameter).

This guide doesn't go into the finer details of the ownership rules in Rust and it is highly recommend to read the documentation provided by the Rust Foundation at `https://doc.rust-lang.org/book/ch04-01-what-is-ownership.html`



Rust Basics

8. Structs & Enums

Compared to other languages like C++, Rust does not offer Classes, however instead it's concepts are found in structs & enums which are used to create custom types in programs. Let's first start of with structs which are common to find in C#, Java, C etc.

8.1 Structs

There are three type of structs that exist in Rust, we have:

1. Empty Structs
2. Tuple Structs
3. Structs

An empty struct is literally what you think, it contains no parameters and tend to be used as a placeholder in programs. A struct can be defined as 'stuct <name>', so an empty struct would look like:

```
struct IamEmpty;
```

A tuple struct is a struct that has no parameters, but instead just asks for the data type, this you can see right below:

```
struct tupleStruct(i32, f32, String);
```

As you can see it looks like a tuple, and doesn't set parameters with actual names. The last one is the classic C struct where parameters and data types are provided, and is the most common type of struct you'll use or encounter.

```
struct classic{
    a: i32,
    b: f32,
    c: String
}

fn main(){
    let c: classic = classic{
        a: 2,
```

```
        b: 7.8,
        c: "Hello".to_owned()
    }; //To invoke a struct
}
```

8.2 Enums

Enums are more custom, as it's not necessary to actually have a data type, of course depending on the context of the program. Let's create a color enum:

```
enum Color{
    Red(u8),
    Green(u8),
    Blue(u8)
}

fn main(){
    let red = Color::Red(255);
    let green = Color::Green(255);
    let blue = Color::Blue(255);
}
```

8.3 Implementations

Of course this is still limiting of what we can do, and that's where implementations come in, these provide functions for our structs or enums. Now let's create a struct for Points and put some functions in it.

```
//To make implementations we need to use the impl keyword
struct Points{
    x: f32,
    y: f32,
}

impl Points{
    fn new (x: f32, y: f32)-> Self {
        //Similar to a constructor
        Self{
            x,
            y
        }
    }
    fn slope(&self, other: &Points)-> f32{
        (other.y - self.y) / (other.x - self.x)
    }
    fn midpoint(p1: &Points, p2: &Points)-> Self{
        Self{
            x: (p1.x + p2.x) / 2.0,
            y: (p1.y + p2.y) / 2.0
        }
    }
}

fn main(){
    let p1 = Points::new(7.8, 8.9);
    let p2 = Points::new(9.8,6.7);

    let midpoint = Points::midpoint(&p1,&p2);
    //Any methods returning Self use ::
```

```
32      //Any methods not returning Self use .
33
34      println!("The midpoint of p1 to p2 is x: {} y: {}", midpoint.x,
        midpoint.y);
35      //The midpoint of p1 to p2 is x: 8.8 y: 7.7999997
36      println!("While the slope of p2 to p1 is {}", p1.slope(&p2));
37      //While the slope of p2 to p1 is -1.0999999
38  }
```