The Rust Programming Language (Reading Notes)

Author: Aslam Ahammed

Source: "The Rust Programming Language" by Nicholas Matsakis and Aaron Turon (It mostly is a living book)

Status: Work in progress

Created using `Typora v0.9.9` as a markdown file.

About me and this notes

I am Aslam Ahammed, a programmer working for a technologies services company, while I'm writing this notes. I took up Rust as a hobby just to ease my eagerness and cure my anxiety. When I say took up Rust, I meant to say started reading and working with the samples given in the book "The Rust Programming Language". The Book, "The Rust Programming Language" or often called "The book" is the authentic reference and tutorial for Rust. When I refer to "The book" or "book" generally I mean about the "The Rust Programming Language" book.

I am creating this notes as part of reading "The book" and trying out Rust code in the book. As well as my side notes. This project is a work in progress, this will have the demo code and notes.

About "The Rust Programming Language"

Language creator: Graydon Hoare

First appeared: in 2010; 8 years ago (source: wikipedia.org)

Rust programming language is a Multi-paradigm: `compiled`, `concurrent`, `functional`, `imperative`, `structured`, `generic` programming language by design.

Rust is Statically typed language. Rust's type system can be classified into `static`, `strong`, `inferred`, `nominal`, as well as `linear`

As of writing, the latest version of Rust language is 1.26.1. And the `rustc` Rust language compiler is compiled in Rust language itself.

Operating system compatibility: Linux, macOS, Windows, FreeBSD, OpenBSD, Redox (operating system), Android, iOS (partial)

Other features includes:

- Compiles to machince code or binary
- Low level access to memory allocation
- Variables are immutable by default, mutable variables cab be declared using `mut` keyword.

And not limited to these.

A bit of history (*cleanup required*)

The language grew out of a personal project started in 2006 by Mozilla employee Graydon Hoare, who stated that the project was possibly named after the rust family of fungi. Mozilla began sponsoring the project in 2009 and announced it in 2010. The same year, work shifted from the initial compiler (written in OCaml) to the self-hosting compilerwritten in Rust. Known as rustc, it successfully compiled itself in 2011. rustc uses LLVM as its back end.

The first numbered pre-alpha release of the Rust compiler occurred in January 2012. Rust 1.0, the first stable release, was released on May 15, 2015. Following 1.0, stable point releases are delivered every six weeks, while features are developed in nightly Rust and then tested with alpha and beta releases that last six weeks.

In addition to conventional static typing, before version 0.4, Rust also supported typestates. The typestate system modeled assertions before and after program statements, through use of a special `check` statement. Discrepancies could be discovered at compile time, rather than when a program was running, as might be the case with assertions in C or C++ code. The typestate concept was not unique to Rust, as it was first introduced in the language NIL. Typestates were removed because in practice they found little use, though the same functionality can still be achieved with branding patterns.

The style of the object system changed considerably within versions 0.2, 0.3 and 0.4 of Rust. Version 0.2 introduced classes for the first time, with version 0.3 adding a number of features including destructors and polymorphism through the use of interfaces. In Rust 0.4, traits were added as a means to provide inheritance; interfaces were unified with traits and removed as a separate feature. Classes were also removed, replaced by a combination of implementations and structured types.

Starting in Rust 0.9 and ending in Rust 0.11, Rust had two built-in pointer types, `~` and `@`, simplifying the core memory model. It reimplemented those pointer types in the standard library as `Box` and (the now removed) `Gc`.

In January 2014, the editor-in-chief of *Dr Dobb's*, Andrew Binstock, commented on Rust's chances to become a competitor to C++, as well as to the other upcoming languages D, Go and Nim (then Nimrod): according to Binstock, while Rust was "widely viewed as a remarkably elegant language", adoption slowed because it repeatedly changed between versions. Rust was the third most loved programming language in the 2015 Stack Overflow annual survey, and took first place in 2016, 2017, and 2018.

The language is referenced in The Book of Mozilla as "oxidised metal."

History source: wikipedia.org

How to install?

Install the compiler and toolchain in linux or mac using the following command

```
$ curl https://sh.rustup.rs -sSf | sh
```

It provides you with different options to install, follow the recommended path or the first option.It will install `rustc`, `rustup` and `cargo`.

- `rustc` - is the compiler
- `rustup` - is the installer for the compiler and toolchain.
- `cargo` - is the package manager and the build tool for Rust projects. Like npm for node.js and pip for python

Updating and uninstalling

To update the toolchain and compiler

```
$ rustup update
```

To uninstall everything

```
$ rustup self uninstall
```

I could find a flavour of `go` in `golang` and `npm` in `node.js` in `cargo`

How the `Hello, World!` look like

Simple one

In the simplest way open a text file in your favorite text editor.

`$ vim hello.rs`

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

Save the file and run `$ rustc hello.rs` if you check the directory where you have the `hello.rs` file you should see an executable named `hello` now.

```
$ ls
hello hello.rs
```

execute the file

```
$ ./hello
Hello, World!
```

here the statement `println!("Hello, World!")` did print to the stdout of the terminal. if you look carefuly there is an `!` after in `println!("Hello, World!")` call. The `!` symbol indicates that this is not a normal function instead it is a macro. Here `println` is a macro.

Using cargo

Create a projects directory and inside that create `hello_cargo` project using `cargo`.

```
$ mkdir projects
$ cd projects
$ cargo new hello_cargo --bin
$ cd hello_cargo
$ ls
Cargo.toml src
$ ls src
main.rs
$ cat src/main.rs
fn main() {
    println!("Hello, world!");
}
$ cargo run
    Finished dev [unoptimized + debuginfo] target(s) in 0.0 secs
     Running `target/debug/hello_cargo`
Hello, world!
```

`cargo new hello_cargo --bin` creates the folder structure, an empty git repo, and Cargo.toml file as well as main.rs file with `main()` and `println!("Hello, world!)`

How to Run?

```
$ cargo run
    Finished dev [unoptimized + debuginfo] target(s) in 0.0 secs
     Running `target/debug/hello_cargo`
Hello, world!
```

How to Build?

There are two ways to build using cargo. The default is to build for debug and for release build we need to pass `--release` flag to `cargo build`

Debug build

```
$ cargo build
   Compiling hello_cargo v0.1.0
(file:///home/aslam/Documents/workspace/rust/projects/hello_cargo)
    Finished dev [unoptimized + debuginfo] target(s) in 0.35 secs
```

Release build

for optimized release build

```
$ cargo build --release
   Compiling hello_cargo v0.1.0
(file:///home/aslam/Documents/workspace/rust/projects/hello_cargo)
    Finished release [optimized] target(s) in 0.34 secs
```

There can be further optimizations, that can be done to the binary output. Which is not present in the book right now. We will discuss that in a later point in time.(***Reminder: Get the Rust blog article to reduce the binary size***)

About the `Cargo.toml` file.

This file is in `toml` format. toml filename extension stands for "Tom's Obvious, Minimal Language" which is a markup language.

This file describes about the package iteself and the dependencies for this rust package(crate). All library packages in rust is called as crates.

sample file content:

```
[package]
name = "hello_cargo"
version = "0.1.0"
authors = ["Aslam Ahammed <aslamplr@gmail.com>"]

[dependencies]
```

The first line `[package]` is a section. And the rest are keys or properties and values. Denoting the name, version and authors. authors is a list of string.

The next section `[dependencies]` is for the dependent crates or packages to this package or crate. In the this sample file content it is left empty as the `hello_cargo` project doesn't have any dependent crates.

A `crate` is a package that is a library, binary packages are not crates **(verify this statement)**

And the file `Cargo.toml` is more like `package.json` file in a node.js project.

About the `Cargo.lock` file.

This file is again a `toml` file without the filename extension though.

Sample file content:

```
[[package]]
name = "hello_cargo"
version = "0.1.0"
```

`Cargo.lock` file is more like `package-lock.json` file in a node.js project.

"Programming a Guessing Game" in Rust

This is in Chapter 2 of "The Book".

This is a CLI based simple number guessing game. This chapter is designed mainly to familiarize with different constructs and common concepts of the Rust programming language. This chapter is more about How to use Rust in a real world program. This chapter introduces the following:

- `let` - keyword that let you define a variable
- `match` - keyword that let you use branching like that of switch.. case construct in other languages.
- methods and traits - trait is similar to a class or a struct and method is similar to classmethod or member function of a class or a struct.
- associated functions - these are more like a static method for a type.
- using external crates - importing the external package or crate into scope.
- etc..

Setting up the project

```
$ cargo new guessing_game --bin
$ cd guessing_game
```

Let's inspect the `Cargo.toml` file.

```
$ cat Cargo.toml
[package]
name = "guessing_game"
version = "0.1.0"
authors = ["Aslam Ahammed <aslamplr@gmail.com>"]

[dependencies]
```

If you inspect the `src/main.rs` file. It will have the `println!("Hello, world!")` program in it. Let us run the `cargo run` and see.

```
$ cargo run
   Compiling guessing_game v0.1.0
(file:///home/aslam/Documents/workspace/rust/projects/guessing_game)
    Finished dev [unoptimized + debuginfo] target(s) in 0.36 secs
     Running `target/debug/guessing_game`
Hello, world!
```

As per the book the `run` command is particularly useful when you need to rapidly iterate on a project.

Processing a guess

Or getting the user input, that is the number guess from the user.

Let's change the content of `src/main.rs` file.

```
use std::io;

fn main() {
    println!("Guess the number!");

    println!("Please input your guess.");

    let mut guess = String::new();

    io::stdin().read_line(&mut guess)
        .expect("Failed to read line");

    println!("You guessed: {}", guess);
}
```

and run it.

```
$ cargo run
   Compiling guessing_game v0.1.0
(file:///home/aslam/Documents/workspace/rust/projects/guessing_game)
    Finished dev [unoptimized + debuginfo] target(s) in 0.44 secs
     Running `target/debug/guessing_game`
Guess the number!
Please input your guess.
1
You guessed: 1
$ cargo run
    Finished dev [unoptimized + debuginfo] target(s) in 0.0 secs
     Running `target/debug/guessing_game`
Guess the number!
Please input your guess.
14
You guessed: 14
```

As you can see, I've run it twice and it was able to capture my input and print it back. I tried with numbers here, let me run one more and try a string as input.

```
$ cargo run
    Finished dev [unoptimized + debuginfo] target(s) in 0.0 secs
     Running `target/debug/guessing_game`
Guess the number!
Please input your guess.
Aslam Ahammed
You guessed: Aslam Ahammed
```

Yes, it works. As it treats the input as a string here in this line `let mut guess = String::new()`. Let's go through line by line of the above program that we have written.

`use std::io` - this is importing from library named `io` inside the `std` library.

From the book: "By default, Rust brings only a few types into the scope of every program in the prelude. If a type you want to use isn't in the prelude, you have to bring that type into scope explicitly with a use statement. Using the std::io library provides you with a number of useful features, including the ability to accept user input."

Then similar to that of in the `Hello, world!` program there is `fn main {` which declares the main entry point function of the program. `()` empty paranthesis means there are no parameters are being passed into this function and `{` starts the body of the function where we can write the Rust statements to be executed and should be closed using `}` in the end.

For the following statements:

```
println!("Guess the number!");

println!("Please input your guess.");
```

As discussed in the `Hello, World!` program. `println!` is a macro. We are yet to read about macros in the following chapters of "The Book".

Storing values with variables

The statement `let mut guess = String::new();`. Here we are defining a variable using the `let` keyword. `mut` keyword is to indicate that this declared variable is "mutable". Since in Rust programming language, default behaviour of a variable is "immutable".

For example:

```
let foo = bar; // foo is immutable here
foo = something_else; // this will throw a compiler error.

let mut foo = bar; // foo is mutable here
foo = something_else; // this will not throw any compiler error.
```

Note: `//` let you write inline comments and `/* ... */` let you write block comments.

That means `let mut guess` will create a mutable variable named guess. And for those in the right hand side of the = sign `String::new();` is the value that variable `guess` is going to be bound to. The statement `String::new()` returns a new instance of a `String` type. `String` is a string type provided by the standard library that is a growable, UTF-8 encoded bit of text. `new` is an "associated function" (static method) of `String` type.

From the book: "The `::` syntax in the `::new` line indicates that new is an associated function of the `String` type. An associated function is implemented on a type, in this case `String`, rather than on a particular instance of a `String`. Some languages call this a static method."

Next,

```
io::stdin().read_line(&mut guess)
    .expect("Failed to read line");
```

If we hadn't listed the `use std::io` line at the beginning of the program, we could have written this function call as `std::io::stdin`.

The `stdin` function returns an instance of `std::io::Stdin`, which is the type that represents a handle to the standard input for the terminal.

Next part `.read_line(&mut guess)`, which calls the `read_line` method on the standard input handle of type `std::io::Stdin` to get the input from the terminal that user has entered. For `&mut guess` that is being passed into `read_line`. The `read_line` will take whatever the user has entered into the terminal and place that into a string. So it takes that string as an argument. The string variable need to be mutable in order for the `read_line` to modify the value of the variable. `&` symbol indicates that the "refrence" should be passed into the method rather than the "value". And the `mut` keyword is needed here since references are immutable by default.

Handling Potential Failure with the `Result` Type

Next part of the statement `.expact("Failed to read line");`. The `read_line` method returns a value of type `io::Result` which is an enumeration type or `enums`. For `Result` enum, the variants are `Ok` or `Err`. The `Ok` variant indicates the operation was successful, and inside `Ok` is the successfully generated value. The `Err` variant means the operation failed, and `Err` contains information about how or why the operation failed. Like any other type `Result` has methods defined in them. An instance of `io::Result` has an expect method that you can call. If this instance of `io::Result` is an `Err` value, expect will cause the program to crash and display the message that you passed as an argument to expect. If the `read_line` method returns an `Err`, it would likely be the result of an error coming from the underlying operating system. If this instance of `io::Result` is an `Ok` value, expect will take the return value that `Ok` is holding and return just that value to you so you can use it. In this case, that value is the number of bytes in what the user entered into standard input.

If you don't call expect, the program will compile, but you'll get a warning:

```
$ cargo build
   Compiling guessing_game v0.1.0
(file:///home/aslam/Documents/workspace/rust/projects/guessing_game)
warning: unused `std::result::Result` which must be used
  --> src/main.rs:10:5
   |
10 |     io::stdin().read_line(&mut guess);
   |     ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
   |
   = note: #[warn(unused_must_use)] on by default

    Finished dev [unoptimized + debuginfo] target(s) in 0.42 secs
```

From the book: "The right way to suppress the warning is to actually write error handling, but since you just want to crash this program when a problem occurs, you can use expect. You'll learn about recovering from errors in Chapter 9."

Missing pieces

Missing topics to be added here from the rest of the Chapter 2, about rand, crates and the rest

The completed program

The entire program of guessing_game:

```
extern crate rand;

use std::io;
use std::cmp::Ordering;
use rand::Rng;

fn main() {
    println!("Guess the number!");

    let secret_number = rand::thread_rng().gen_range(1, 101);

    loop {
        println!("Please input your guess.");

        let mut guess = String::new();

        io::stdin().read_line(&mut guess)
            .expect("Failed to read line");

        let guess: u32 = match guess.trim().parse() {
            Ok(num) => num,
            Err(_) => continue,
```

```
        };

        println!("You guessed: {}", guess);

        match guess.cmp(&secret_number) {
            Ordering::Less => println!("Too small!"),
            Ordering::Greater => println!("Too big!"),
            Ordering::Equal => {
                println!("You win!");
                break;
            }
        }
    }
}
```

Summary

Summerize here