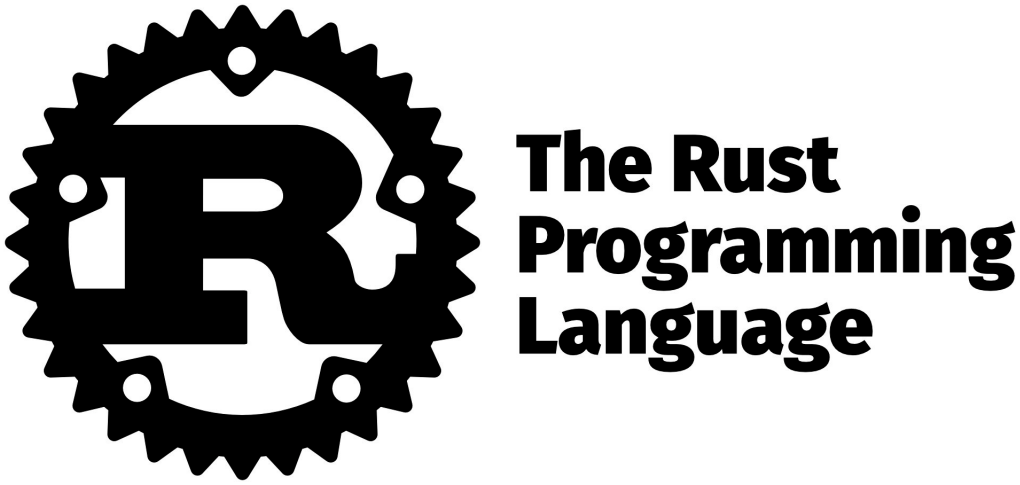
Moving to the Rust programming language

Jaideep Ganguly, ScD



Contents

List of Tables

List of Listings

Preface

Rust is an open-source, community-developed systems programming language that runs blazing fast, prevents segfaults, and guarantees thread safety.

The motivation to write this book came from the desire to develop a concise and yet a comprehensive for a rigorous exposure to Rust. The goal is to help proficient developers in well known languages such as Java, C#, Kotlin, Scala, Go, etc., to quickly migrate to Rust.

The official Rust book is excellent but at nearly 400 pages it is rather verbose. It can be followed even by developers who have less exposure to programming. Consequently, the authors have gone to a great length to write a great book lucid with examples.

For developers proficient in some other language, this book will help them migrate to Rust in just a couple of days. The content has been deliberately limited to just about a hundred pages.

The examples used in the book can be found in github at:
https://github.com/jganguly/rustfun

Jaideep Ganguly received his degrees of Doctor of Science and Master of Science from the Massachusetts Institute of Technology. He received his undergraduate degree from the Indian Institute of Technology, Kharagpur. He started his career at Oracle and has worked with Microsoft, Amazon and is now heading the India R&D center of compass.com.

CHAPTER 1

WHY RUST?

The Rust programming language has been Stack Overflow's most loved language for five years in a row, clearly establishing the fact that a significant section of the developer population love it. Rust solves many pain points present in current popular languages and has a limited number of downsides.

It's quite difficult to write secure code for large programs. It's particularly difficult to manage memory correctly in C and C++. As a result we see a regular procession of security breaches starting from the Morris worm of 1988. Furthermore, It's even more difficult to write multi threaded code, which is the only way to exploit the abilities of modern multi CPU machines. Concurrency can introduce broad new classes of bugs and make ordinary bugs much harder to reproduce.

Rust has been designed to be a safe, concurrent language and yet match or better the performance of C/C++. Rust is not really an object-oriented language, although it has some characteristics of it. It is also not a functional language, but does adhere to many the many tenets of functional programming.

although it does tend to make the influences on a computation's result more explicit, as functional languages do.

1.1 STATICALLY TYPED

Rust is a statically and strongly typed systems programming language. Statically means that all types are known at compile-time, strongly means that these types are designed to make it harder to write incorrect programs. A successful compilation means you have a much better guarantee of correctness than with language such as C or Java, generating the best possible machine code with full control of memory use.

The debate between dynamic versus static typed is long standing and will continue. However, dealing with dynamic typing in large code bases is difficult. Statically typed languages allow the compiler to check for constraints on the data and its behavior and thereby significantly reducing cognitive overheads on the developer to produce correct code. Statically typed languages differ from each other. An important aspect is how they deal with the concept of NULL, which means the value may be something or nothing. Like Haskell and some other modern programming languages, Rust encodes this possibility using an optional type and the compiler requires you to handle the None case.

1.2 TYPE SAFETY

The basis for computers and computer programs is the Von Neumann model which is over seventy years old. This architecture has one memory that holds both the instructions as well as program data. The commonly used programming languages C and C++ offer no support

for automatic memory management. The programmer has to deal with memory management making the program error prone. A common source of error is the well known *buffer overflow*. Buffer overruns are the cause of many computer problems, such as crashing and vulnerability in terms of attacks. The past years have revealed multiple exploits: vulnerabilities that are the result of these types of errors and which could have significant consequences.

Recently, a serious vulnerability known as *Heartbleed* was discovered. The bug was in *OpenSSL* which is commonly used in network routers and web servers. This vulnerability allowed encryption keys to be read remotely from these systems and thereby compromised their security. This vulnerability would have been avoided Rust was used to develop OpenSSL.

Rust is safe by default. All memory accesses are checked and It is not possible to corrupt memory by accident. With direct access to hardware and memory, Rust is an ideal language for embedded and bare-metal development. One can write extremely low-level code, such as operating system kernels or micro-controller applications. However, it is also a very pleasant language to write application code as well. Rust's core types and functions as well as reusable library code stand out in these especially challenging environments. However, unlike many existing systems programming languages, Rust does not require developers to spend their time mired in nitty-gritty details.

Rust's strong type system and emphasis on memory safety, all enforced at compile time, mean that it is extremely common to get errors when compiling your code. This can be a frustrating feeling for programmers not used to such an opinionated programming language. However, the Rust developers have spent a large amount of time working to improve the error messages to ensure that they are clear and actionable. One must not gloss over Rust compile time error messages.

In summary, if a program has been written so that no possible execution can exhibit undefined behavior, we say that program is well defined. If a language's safety checks ensure that every program is well defined, we say that language is type safe.

1.3 RUNTIME

Rust strives to have as many zero-cost abstractions as possible, abstractions that are as equally performant as corresponding hand-written code. Zero-cost abstraction means that there's no extra runtime overhead that you pay for certain powerful abstractions or safety features that you do have to pay a runtime cost for other languages. However, be aware that not every abstraction or every safety feature in Rust is truly *zero-cost*.

A programming language runtime, is all the machinery provided by the language itself and which is injected into and supports the execution environment. This can include things as small as some minimal routines for laying out and freeing memory, as in C, to entire virtual machines, interpreters, and standard libraries, as in Java, Python or Ruby. Think of it as the minimal machinery that is both part of the language and must be present, either in the executable itself or installed on the computer, for any given program written in that language to run.

Rust strives to have a very fast run time. It does this in part by compiling to an executable and injecting only a very minimal language runtime and *does not provide a memory manager, i.e., a garbage collector* that operates during the executable's runtime.

1.4 PERFORMANCE

Rust gives you the choice of storing data on the stack or on the heap and determines at compile time when memory is no longer needed and can be cleaned up. This allows efficient usage of memory as well as more performant memory access. Tilde, an early production user of Rust in their Skylight product, found they were able to reduce their memory usage from 5GB to 50MB by rewriting certain Java HTTP endpoints in idiomatic Rust. Savings like this quickly add up when cloud service providers charge premium prices for increased memory or additional machines.

Without the need to have a garbage collector continuously running, Rust projects are well-suited to be used as libraries by other programming languages via foreign-function interfaces. This allows existing projects to replace performance critical pieces with speedy Rust code without the memory safety risks inherent with other systems programming languages. Some projects are being incrementally rewritten in Rust using these techniques.

The fundamental principles of Rust are:

1. *ownership* and safe *borrowing* of data
2. *functions*, *methods* and *closures* to operate on data
3. *tuples*, *structs* and *enums* to aggregate data
4. *matching* pattern to select and destructure data
5. *traits* to define behavior on data

CHAPTER 2

GETTING STARTED

2.1 INSTALLATION

In MacOS or Linux, Rust is installed using the following command in the terminal.

```
# install rust
curl --proto '=https' --tlsv1.2 https://sh.rustup.rs -sSf | sh

# update rust
rustup update

# uninstall rust
rustup self uninstall

# rust compiler
rustc --version
```

Listing 2.1: Installation in MacOS or Linux

2.2 CREATING PROJECT WITH CARGO

`Cargo` is Rust's build system and package manager. For simple projects, you can use the `rustc` compiler but for complex projects you have to use `Cargo`. It is used to manage Rust projects because `Cargo` handles a lot of tasks such as building the code, downloading the libraries the code depends on and building those libraries. `Cargo` comes installed with Rust and you can check whether it is installed by typing the following in a terminal.

```
cargo --version
```

Listing 2.2: Cargo

Let us create a directory **my_rust_project** to store the Rust code in your home directory.

```
cargo new my_rust_project
cd my_rust_project
```

Listing 2.3: Project with Cargo

The first command creates a new directory called **my_rust_project** in your home directory. We've named our project **my_rust_project**, and Cargo creates its files in a directory of the same name. Go into the **my_rust_project** directory and list the files. You'll see that Cargo has generated two files and one directory for us: a Cargo.toml file and a src directory with a main.rs file inside.

It has also initialized a new Git repository along with a .gitignore file. Git files won't be generated if you run cargo new within an existing Git repository; you can override this behavior by using `cargo new --vcs=git`.

The contents of Cargo.toml look as follows:

```
[package]
name = "tpl"
version = "0.1.0"
authors = ["Jaideep Ganguly <ganguly.jaideep@gmail.com>"]
edition = "2018"

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

[dependencies]
```

Listing 2.4: Project with Cargo

This file is as per the TOML (Tom's Obvious, Minimal Language) format, which is Cargo's configuration format. The first line, `[package]`, is a section heading that indicates that the following statements are configuring a package. The next four lines set the configuration information Cargo needs to compile your program: the name, the version, who wrote it, and the edition of Rust to use. The last line, `[dependencies]`, is the start of a section for you to list any of your project's dependencies. **Packages of code are referred to as crates**.

The file `src/main.rs` has been generated by Cargo.

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

Listing 2.5: Hello World!

Rust files end with the `.rs` extension and `fn` is the keyword to denote a function. Cargo placed the code in the `src` directory and we have a Cargo.toml configuration file in the top directory. Cargo expects your source files to live inside the `src` directory. The top level project directory is just for README files, license information, configuration files, etc. Using Cargo helps you organize your projects.

2.3 BUILD & RUN WITH CARGO

To build and run, you need to `cd` to `my_rust_project`.

```
cargo build # build only
cargo run   # build and run
```

Listing 2.6: Build & Run

`cargo build` command creates an executable file in `target/debug/tpl`. We can run it with `./target/debug/hello_world`. Running cargo build for the first time also causes Cargo to create a new file at the top level Cargo.lock. This file keeps track of the exact versions of dependencies in your project. We can also use cargo run to compile the code and then run the resulting executable all in one command. Cargo also provides a command called

`cargo check`. that checks your code to make sure it compiles but doesn't produce an executable. Cargo check is much faster than cargo build, because it skips the step of producing an executable. If you're continually checking your work while writing the code, using cargo check will speed up the process. When your project is finally ready for release, you can use `cargo build --release` to compile it with optimizations. This command will create an executable in target/release instead of target/debug. The optimizations make your Rust code run faster, but turning them on lengthens the time it takes for your program to compile.

2.4 CREATING A LIBRARY WITH CARGO

Cargo is used to create a library named `tpllib` using the command:
`cargo new --lib my_rust_lib` to create the library `my_rust_lib`.

We will learn more about how to structure your code in separate modules and libraries.

Bibliography

[1] Steve Klabnik and Carol Nichols, with contributions from the Rust Community, *The Rust Programming Language*. https://doc.rust-lang.org/book, 2018.

[2] Donald E. Knuth, *The Art of Computer Programming*. Addison-Wesley, 2011.

[3] Alfred V. Aho, Monica S. Lam, Ravi Shethi , Jeffery D. Ullman, *Compilers: Principles, Techniques, and Tools*. Prentice Hall, 2006.

[4] Kanglin Li, Mengqi Wu, Sybex, *Effective Software Test Automation: Developing an Automated Software Testing Tool*. Sybex , 2004.

[5] Martin Odersky, *The Scala Language Specification Version 2.9* PROGRAMMING METHODS LABORATORY EPFL, SWITZERLAND, 2014.