Mastering Rust: A Comprehensive Guide
- From Basics to Advanced Patterns and Best Practices -

Ji Yong-Hyeon

Department of Information Security, Cryptology, and Mathematics
College of Science and Technology
Kookmin University

March 4, 2024

Rust vs C

Memory Safety
One of Rust's core features is its guarantee of memory safety without a garbage collector. Rust achieves this through its ownership system, consisting of three main rules:

- **Ownership**: Each value in Rust has a single owner, the variable that binds to it.
- **Borrowing**: You can have references to a value, but these must adhere to borrowing rules. At any given time, you can have either one mutable reference or any number of immutable references to a particular piece of data.
- **Lifetimes**: The compiler uses lifetimes to ensure that references do not outlive the data they refer to. This system prevents common bugs found in C programs, such as dangling pointers, buffer overflows, and concurrent data races.

Concurrency
Rust's approach to concurrency is also built on the ownership and type system, enabling thread-safe programming without the fear of data races. This is a significant shift from C, where managing concurrent access to data can be error-prone and complex.

Error Handling
Rust encourages a more explicit approach to error handling compared to C's use of integers for indicating errors. Rust uses Result and Option types for functions that can fail or return nothing, respectively, forcing you to handle these cases explicitly and improving code reliability.

Package Management and Build System
Rust comes with Cargo, its package manager and build system, simplifying dependency management, compilation, and packaging. Think of it as combining make, gcc, and a package repository like npm or Maven. The Cargo.toml file (which you asked about) is part of this system, defining your project's dependencies, build configurations, and metadata.

Zero-Cost Abstractions
Rust provides high-level abstractions like iterators, closures, and pattern matching without sacrificing performance. These features enable expressive and efficient code but compile down to low-level code with performance equivalent to C.

Compatibility with C
Rust can interoperate with C by using foreign function interfaces (FFI). This means you can call C code from Rust and vice versa, allowing gradual migration of C codebases to Rust or leveraging existing C libraries.

Learning Curve
While Rust offers many benefits, it has a steeper learning curve than C, primarily due to its strict compiler checks, ownership model, and borrowing rules. However, these same features that make Rust challenging to learn also contribute to its safety and efficiency.

Summary As a C developer, you'll find familiar ground in Rust's performance and control but with added safety and modern features. The ownership model is the most novel aspect, eliminating entire classes of bugs prevalent in C programs. Transitioning to Rust will involve a shift in thinking, particularly around memory management and type safety, but it can lead to more reliable and robust system-level applications.

Contents

1 Introduction to Rust Part I **1**
- 1.1 Setting Up and Getting Started 1
  - 1.1.1 Installing Rust (Linux) 1
  - 1.1.2 Rust Terminal 2
  - 1.1.3 VSCode 4
- 1.2 Basic Rust Development 5
  - 1.2.1 Project Overview 5
  - 1.2.2 Cargo Run 5
  - 1.2.3 Creating a File 5
- 1.3 title 6

Chapter 1

Introduction to Rust Part I

1.1 Setting Up and Getting Started

1.1.1 Installing Rust (Linux)

"How to install Rust and the necessary tools on your system"

1.1.2 Rust Terminal

"Introduction to using the Rust terminal for compiling and running Rust programs"

```
@:~$ rustup
rustup 1.26.0 (5af9b9484 2023-04-05)
The Rust toolchain installer
```

Table 1.1: Rustup 1.26.0 Command-Line Interface Documentation

Section	Content
Usage	`rustup [OPTIONS] [+toolchain] <SUBCOMMAND>`
Arguments	`<+toolchain>` - Release channel (e.g., +stable) or custom toolchain to set override.
Options	`-v, -verbose` - Enable verbose output. `-q, -quiet` - Disable progress output. `-h, -help` - Print help information. `-V, -version` - Print version information.
Subcommands	`show` - Show the active and installed toolchains or profiles. `update` - Update Rust toolchains and rustup. `check` - Check for updates to Rust toolchains and rustup. `default` - Set the default toolchain. `toolchain` - Modify or query the installed toolchains. `target` - Modify a toolchain's supported targets. `component` - Modify a toolchain's installed components. `override` - Modify directory toolchain overrides. `run` - Run a command with an environment configured for a given toolchain. `which` - Display which binary will be run for a given command. `doc` - Open the documentation for the current toolchain. `man` - View the man page for a given command. `self` - Modify the rustup installation. `set` - Alter rustup settings. `completions` - Generate tab-completion scripts for your shell. `help` - Print this message or the help of the given subcommand(s).
Discussion	Rustup installs The Rust Programming Language from the official release channels, enabling you to easily switch between stable, beta, and nightly compilers and keep them updated. It makes cross-compiling simpler with binary builds of the standard library for common platforms. If you are new to Rust, consider running `rustup doc --book` to learn Rust.

```
@:~$ cargo
Rust`s package manager
```

Table 1.2: Cargo Command-Line Interface

Section	**Content**

Usage	`cargo [+toolchain] [OPTIONS] [COMMAND]`
	`cargo [+toolchain] [OPTIONS] -Zscript <MANIFEST_RS> [ARGS]...`

Options	`-V, -version` - Print version info and exit. `-list` - List installed commands. `-explain <CODE>` - Provide a detailed explanation of a rustc error message. `-v, -verbose...` - Use verbose output (-vv very verbose/build.rs output). `-q, -quiet` - Do not print cargo log messages. `-color <WHEN>` - Coloring: auto, always, never. `-C <DIRECTORY>` - Change to DIRECTORY before doing anything (nightly-only). `-frozen` - Require Cargo.lock and cache are up to date. `-locked` - Require Cargo.lock is up to date. `-offline` - Run without accessing the network. `-config <KEY=VALUE>` - Override a configuration value. `-Z <FLAG>` - Unstable (nightly-only) flags to Cargo, see 'cargo -Z help' for details. `-h, -help` - Print help.
Commands	`build, b` - Compile the current package. `check, c` - Analyze the current package and report errors, but don't build object files. `clean` - Remove the target directory. `doc, d` - Build this package's and its dependencies' documentation. `new` - Create a new cargo package. `init` - Create a new cargo package in an existing directory. `add` - Add dependencies to a manifest file. `remove` - Remove dependencies from a manifest file. `run, r` - Run a binary or example of the local package. `test, t` - Run the tests. `bench` - Run the benchmarks. `update` - Update dependencies listed in Cargo.lock. `search` - Search registry for crates. `publish` - Package and upload this package to the registry. `install` - Install a Rust binary. Default location is $HOME/.cargo/bin. `uninstall` - Uninstall a Rust binary. `...` - See all commands with –list.
Additional Information	See '`cargo help <command>`' for more information on a specific command.

1.1.3 VSCode

“Setting up Visual Studio Code for Rust development, including recommended extensions and configurations ”

1.2 Basic Rust Development

1.2.1 Project Overview

"Understanding the structure of a Rust project and the purpose of each file"

```
@~$ cargo new my_project
Created binary (application) `my_project` package
```

```
@:~$ cd my_project
@:~$ tree
.
¦-- Cargo.toml
'-- src
        '-- main.rs

1 directory, 2 files
```

```
@:~$ cat Cargo.toml
[package]
name = "my_project"
version = "0.1.0"
edition = "2021"

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

[dependencies]
```

1.2.2 Cargo Run

"How to build and run Rust projects using Cargo, Rust's package manager and build tool"

1.2.3 Creating a File

"Basics of file operations in Rust, including creating new files"

1.3 title

Rust Programming Fundamentals:

- **C:**

  `char, int, float, double`

- **Rust:**

  `u8, u32, u64, i8, i32, i64, f32, f64`

```rust
fn main() {
  // This is a comment
  println!("Hello, world!"); // Print "Hello, world!" to the console
}
```

```rust
fn main() {
  let value_1: i32 = 50;
  let value_2: f64 = 3.14;
  let result: value_1 + value_2 as i32;
  println!("{}", result);
}

fn main() {
  let value_1: i32 = 50;
  let value_2: f64 = 3.14;
  let result: value_1 as f64 + value_2;
  println!("{}", result);
}
```

Bibliography

[1] “Introduction to Rust Part 1” YouTube, uploaded by Ryan Levick, 16 November 2020, https://www.youtube.com/watch?v=WnWGO-tLtLA