Draft. Please do not include this page.

Single chapter compiled via LaTeX.

Shing Lyu, June 18, 2022. Git commit: 65314af

Notes for the production team

Whitespace marker

You will see pink underline markers like these: `a␣b␣c`. One marker indicates one whitespace. So there is exactly one whitespace between "a" and "b" and one whitespace between "b" and "c". You'll also see them in code blocks like this:

```
#[derive(StructOpt)]
struct␣Options␣{
␣␣␣␣//␣...
␣␣␣␣#[structopt(short␣=␣"f",␣long␣=␣"file",␣parse(from_os_str))]
␣␣␣␣///␣Load␣the␣cat␣picture␣from␣the␣specified␣file
␣␣␣␣catfile:␣Option<std::path::PathBuf>,
}
```

These markers are only for you to see how many whitespaces there are. Please keep the exact number of whitespaces for inline code and code blocks. **Please do NOT include the markers during layout**.

The layout output should look like this: `a b c`

```
#[derive(StructOpt)]
struct Options {
    // ...
    #[structopt(short = "f", long = "file", parse(from_os_str))]
    /// Load the cat picture from the specified file
    catfile: Option<std::path::PathBuf>,
}
```

Auto line wrapping marker

You'll also see arrows near the end of the line and beginning of the second line:

```
This is a very very very very very very very very very very ↵
    ↳very very very long line that was wrapped automatically.
```

This indicates the original line was too long and it was automatically wrapped. **Please include these arrows.**

CHAPTER 1

Welcome to the world of Rust

If you are reading this book, you are probably as excited about Rust just as I. Ever since the first stable release in 2015, Rust has come a long way in terms of features and stability. Developers around the world are fascinated about how Rust can combine features that were once thought as unavoidable tradeoffs: performance with memory safety, low-level control with productivity. Despite the infamous steep learning curve, Rust has gained popularity over the years. It was named the "most loved programming language" in the StackOverflow survey four years in a row from 2016 to 2019. Many big companies and organizations like Facebook, Microsoft, Dropbox, and npm also have started to use Rust in production.

What are people using Rust for? If we take a look at crates.io, the official Rust crates (libraries) registry, there are over 28,900 crates and over a billion downloads. There are 47 categories on crates.io[1], ranging from the command-line interface, cryptography, database, games, operating systems, web programming, and many more. What does it feel like to use these libraries? How do Rust's syntax and design philosophy affect the design of these crates? This book will try to answer these questions.

1.1 Who is this book for?

This book will be useful for:

- People who already know basic Rust syntax, but want to learn how to build applications in Rust.
- People who are considering using Rust to build production-ready systems.
- People who wish to have a quick overview of high-level architecture and programming interface design in other fields.

You might have learned Rust out of curiosity. After finishing all the tutorials and beginner books, you might be wondering, "What should I do next? What can I build with Rust?" This book will walk you through a few different applications of Rust, which will help you move beyond theories and into building real applications. Rust has a fascinating design and many interesting language features, but simply knowing how to write basic algorithms and data structures won't necessarily prepare you for building useful applications. I tried

[1] https://crates.io/categories

to find the most production-ready but also modern Rust libraries to do the job, so you'll be able to judge if Rust is ready for the application you have envisioned. If it's not, you might even find opportunities to contribute back to Rust's community by improving the existing libraries, frameworks, and design or build new ones.

You might have learned Rust for a specific project, maybe a CLI tool for work or an open-source browser engine that happens to use Rust. Once you master Rust for that domain, it's beneficial to learn Rust for other domains, say building a game or website. This will bring you unexpected ideas. For example, by building a game, you learned how game engine designers organize their code to make it decoupled and easy to maintain. You may never build a game for work, but that architectural knowledge might influence the architecture of your next project. Another example is that learning how cross-compiling to a Raspberry Pi might help you understand how compiling to WebAssembly works. So this book is aimed to take you through a tour of various applications of Rust. You'll learn how their application programming interface (API)[2] looks like and how they organize their code and architecture.

1.2 Who is this book not for?

This book might not be that useful for:

- People who want to learn the Rust programming language itself.
- People who want to dive deep into one particular field.
- People who are looking for the most experimental and cutting-edge Rust implementations.

This book is not a course on the Rust programming language itself, nor is it trying to teach Rust's syntax via examples. We'll focus on the applications themselves and their domain knowledge, assuming you already know Rust's syntax and language features. There are many excellent books on Rust itself, like *The Rust Programming Language* by Steve Klabnik and Carol Nichols. You can also find online books, interactive tutorials, and videos on the *Learn Rust* section of the official website[3]. Each of the topics in this book can easily be expanded into a book on its own, so I will try to give you a high-level understanding of the topic, but won't go too deep into them. I aim to give you a broad overview of what is possible with Rust and how the developer experience is like. Therefore, the examples are simplified so people without a background in that particular field can still get a taste of the mental model of the field. Also, I'm not advocating that the methods presented in the book are the "best" or most trendy way of building those applications. I tried to strike a balance between being modern and being pragmatic.

[2] I use the term API in a general sense. This includes the functions, structs, and command-line tools exposed by each library or framework.

[3] https://www.rust-lang.org/learn

1.3 Criteria for selecting libraries

Rust is a relatively young language. Therefore it's a big challenge to select the libraries or frameworks to use in each chapter. On the one end, there are experimental pure-Rust implementations for almost every field. Many proof-of-concept libraries are competing with each other without a clear winner. The early adopters of Rust are usually adventurous developers; they are comfortable with rough edges in the libraries and find workarounds. The focus is on experimentation, learning, and innovation, but not necessarily user-friendliness. On the other end, the people seeking stability and production-readiness. Because of the great Interoperability of Rust with other programming languages, there are many attempts to write a Rust wrapper around mature C/C++ (or other languages) libraries. In this book, I'm trying to demonstrate the core concept in each field, and what their Rust API design looks like. Therefore, I select the libraries we use the following criteria:

Pure-Rust

I try to find libraries that are built purely in Rust. Rust's FFI (foreign function interface) allows you to call existing C libraries (and many other languages) from Rust. Therefore, the easiest way to build Rust applications quickly is to leverage exiting libraries in other languages. These libraries are usually designed with other languages in mind, so wrapping them in Rust results in a weird and not idiomatic Rust API. So if there is a purely Rust library or library using existing technology but built from scratch using Rust, I tend to choose those.

Maturity

However, not every purely Rust library is very mature. Because many Rust libraries are built from a clean slate, the developers tried to experiment with the latest experimental technology, but that might mean that the architecture and API design is very experimental and changes frequently. Some of the libraries showed great potential in their early days, but then the development slows down, and the projects are eventually going into maintenance mode or even abandoned. We aim to build useful software rather than experiment with exciting technologies and throw the code away. Therefore, we need to be pragmatic and choose a library that is mature enough and uses widely-accepted design patterns, rather than being dogmatic about using pure-Rust libraries. I choose to use a `GTK+`-based library in Chapter 3 for this reason.

Popularity

If two or more candidates are passing the above criteria, I'll choose the most popular one. The popularity is based on a combination of factors like:

- Number of downloads on crates.io.
- Pace of development and release
- Discussions on issue trackers and discussion forums
- Media coverage

Although popularity is not a guarantee to success, a popular project is more likely to have a big enough community the supports it and keeps it alive. This can help us find a library that has the most potential to stick around longer in the future. You are also more likely to get support and answers online.

1.4 How to use this book

The chapters in this book are independent of each other so that you can read them in any order you want. However, some of the ideas or design patterns are used in multiple chapters. I try to introduce these ideas in the chapter where the design originated, or where they make the most sense. For example, the concept of using event handlers to build a responsive user interface is introduced in the Text-based User Interface section in Chapter 3, and then referenced in Chapter 7. So reading the book from cover to cover might help you build up this knowledge in an incremental way.

Chapters overview

In Chapter 2, I started with the easiest application we can build with Rust: a command-line interface (CLI). Building a CLI requires very minimal setup and background knowledge but can produce very powerful applications. I first introduce how to read raw arguments using the standard library, then we show how to use `StructOpt` to manage arguments better, and get features like generating a help message for free. I also touch upon topics like piping, testing, and publishing the program to crates.io.

In Chapter 3, we build two-dimensional interfaces. We first build a text-based 2D interface using the `Cursive` text-based user interface system. This allows us to build interactive user interfaces like popups, buttons, and forms. The experience in the text-based user interface (TUI) paved the way for a graphical user interface (GUI). I'll be introducing the Rust binding for the popular GTK+ 3 library, `gtk-rs`. We build the same interactive form using the GUI library.

In Chapter 7, we'll be building a game in Rust. We use the Amethyst game engine to make a cat volleyball game. You'll learn the design philosophy behind Amethyst called the Entity-Component-System. You'll learn how to create 2D games, rendering the

characters and items with a spritesheet. We'll also implement game logic like collision detection, keeping score, and adding sound effects and background music.

In Chapter 8, we'll connect the virtual world with the physical world. I'll introduce physical computing on a Raspberry Pi development board. We'll start by installing a full operating system and install the whole Rust toolchain on the device, and introduce how to use Rust to control an LED and how to take inputs from a physical button. Then we'll show you how to cross-compile Rust on another computer to produce a binary that runs on a Raspberry Pi.

In Chapter 9, we shift our focus to artificial intelligence and machine learning. I'll show how to implement an unsupervised and supervised machine learning model using the `rusty-machine` crate. For the unsupervised model, we'll be introducing K-means, and for the supervised model, we'll demonstrate the neural network. I'll also be showing how to do some data processing tasks like test data generation, reading/writing CSV files, and visualization.

Finally, in Chapter 10, I give a broad overview of other exciting fields in Rust that can't make it into this book. I'll point you to cool projects in areas like operating systems, web browsers, web servers backend, serverless, frontend (WebExtension). This chapter will act as a guide book for your future exploration into the vast world of Rust.

1.5 Source Code

All the source code for this book is available on GitHub: https://www.github.com/apress/practical-rust-projects. The source code is also accessible via the Download Source Code button located at www.apress.com/9781484255988.

When I include source code in the book, I only highlight the part that is relevant to the point being discussing. The non-relevant part will be omitted with a comment like this:

```
// ...
```

Therefore, not all code examples can be compiled successfully. To check the fully working example, check the source code on GitHub.

Most of the examples are developed and tested on a Linux (Ubuntu 16.04) machine. The Rust version is `stable-x86_64-unknown-linux-gnu unchanged - rustc 1.39.0 (4560ea788 2019-11-04)`. The stable version is used as much as possible. But certain libraries might require us to use the nightly version.