

Vigneshwer Dhinakaran

Rust Cookbook

Learn to build blazingly fast and safe real-world applications using the Rust programming language



Packt

Rust Cookbook

Learn to build blazingly fast and safe real-world applications using the Rust programming language

Vigneshwer Dhinakaran



BIRMINGHAM - MUMBAI

```
<html PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN"  
"http://www.w3.org/TR/REC-html40/loose.dtd">
```


Rust Cookbook

Copyright © 2017 Packt Publishing

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 Packt Publishing, and its dealers and distributors will be held liable for any damages caused or alleged to be caused directly or indirectly by this book.

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

First published: July 2017

Production reference: 1260717

Published by Packt Publishing Ltd.

Livery Place
35 Livery Street
Birmingham
B3 2PB, UK.

ISBN 978-1-78588-025-4

www.packtpub.com

Credits

Author	Copy Editor
Vigneshwer Dhinakaran	Muktikant Garimella Gladson Monteiro
Reviewer	Project Coordinator
Pradeep R	Ulhas Kambali
Commissioning Editor	Proofreader
Aaron Lazar	Safis Editing
Acquisition Editor	Indexer
Denim Pinto	Rekha Nair
Content Development Editor	Graphics
Vikas Tiwari	Abhinash Sahu
Technical Editor	Production Coordinator
Diwakar Shukla	Melwyn Dsa

About the Author

Vigneshwer Dhinakaran is an innovative data scientist with an artistic perception of technology and business, having over 3 years of experience in various domains, such as IOT, DevOps, computer vision, and deep learning, and is currently engaged as a research analyst crunching real-time data and implementing state-of-the-art AI algorithms in the innovation and development lab of the world's largest decision science company in Bengaluru, India.

He is an official Mozilla representative and Techspeaker in India and has been associated with Mozilla communities and technologies for more than 5 years. He has delivered various sessions on Rust language at many meetups and conferences; some of the highlighted events are Hong Kong Open Source Conference, FOSSMeet 17, and RainOfRust Campaign. He has played a key role in the formation and growth of the Rust community in India and was part of the Mozilla Reps mobilizer experiment, where he researched about the roadblocks and solutions to drive developers to adopt the Rust language in India.

Acknowledgments

I would like to thank the entire Packt team for providing me with all their support and guidance throughout publishing my first book. I would like to call out Mr.Vikas Tiwari, who is the content development editor of the book, for patiently handling all my queries and planning the entire project; it has been a great experience working with you.

Thanks to my college, Rajagiri School of Engineering and Technology, Kochi, from where I got my bachelor's degree. This place provided me with a great platform to learn, explore, and practice various engineering and leadership skills. I would like to express my sincere thanks to my professor, Dr. Deepti Das Krishna, who has always guided me in the right direction and supported all my activities. Thanks to all my teachers who have taught me so many important lessons throughout my life.

Thanks to Mr. Pradeep Ramalingam, the technical reviewer of the book and my mentor for providing support, offering comments, and assisting me during the entire editing, proofreading, and design of the book.

Thanks to Mr. Nitish Bhardwaj, who has always offered me a lot of mentorship and guidance to develop as an engineering professional in the early days of my career.

Being part of the Mozilla community is one the best things that has happened to me. I wouldn't have been here without the amazing exposure and learning experience from the network. I would like to express my gratitude to the many people who have helped me grow as a contributor in the ecosystem, and special thanks to the Mozilla Reps, Mozilla Tech Speakers, and Mozilla India community volunteers. Thanks also to the Rust community team and members for their amazing work and contribution in providing developers with great documentation, and inclusive practices and programs; it has truly been a great learning experience being part of the community.

Thanks to my longtime friends, Harry Prince, Jairam Sankar, Srinivas Srikanth, Sriram Subramanian, Krishna Prasad, Firoz Jamal, and Ashique MN, and to all well wishers for their support and friendship.

Above all, I want to thank my parents, D.Dhinakaran and D.Selvi, for providing me with all the right resources and for being a great source of encouragement and positivity in my life, to my younger sister D.Abhinaya for showing great confidence in my works, and to the rest of my family for their unconditional love and support, inspite of all the time it took me away from them. It was a long and difficult journey for them.

Last but not least, I beg forgiveness of all those who have been with me over the course of the years and whose names I have failed to mention.

About the Reviewer

Pradeep R is an avid programmer who is passionate in working on network security. He is an experienced developer working on network security and network monitoring/visibility platforms devices for the past 5 years. He has worked on areas such as NAT, Firewall, VPN, Intrusion Detection Systems, network switching, and routing solutions.

He loves different programming languages and strongly believes that all programming languages are similar in essence and can be easily adapted. His area of interest spans over different programming languages and extensively works with C, C++, Python, JavaScript, Perl, and occasionally with Java, .NET, and Rust.

He is currently working as a lead engineer in Gigamon Inc. on network visibility devices. The Gigamon Inc. manufactures network visibility next generation devices that are used for analyzing the network traffic and monitors them to detect malicious activity or determine abnormal network usage pattern to detect security breach.

www.PacktPub.com

For support files and downloads related to your book, please visit www.PacktPub.com. Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at www.PacktPub.com and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at service@packtpub.com for more details. At www.PacktPub.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 Packt books and eBooks.



<https://www.packtpub.com/mapt>

Get the most in-demand software skills with Mapt. Mapt gives you full access to all Packt books and video courses, as well as industry-leading tools to help you plan your personal development and advance your career.

Why subscribe?

- Fully searchable across every book published by Packt
- Copy and paste, print, and bookmark content
- On demand and accessible via a web browser

Customer Feedback

Thanks for purchasing this Packt book. At Packt, quality is at the heart of our editorial process. To help us improve, please leave us an honest review on this book's Amazon page at <https://www.amazon.com/dp/178588025X>. If you'd like to join our team of regular reviewers, you can e-mail us at customerreviews@packtpub.com. We award our regular reviewers with free eBooks and videos in exchange for their valuable feedback. Help us be relentless in improving our products!

Table of Contents

Preface

What this book covers

What you need for this book

Who this book is for

Sections

 Getting ready

 How to do it...

 How it works...

 There's more...

 See also

Conventions

Reader feedback

Customer support

 Downloading the example code

 Downloading the color images of this book

 Errata

 Piracy

 Questions

1. Let us Make System Programming Great Again

 Introduction

 Setting up Rust in Linux and macOS

 Getting ready

 How to do it...

 Uninstalling Rust

 Rust's compiler version

 Advanced installation options

 Troubleshooting

 How it works...

 Setting up Rust in Windows

 Getting ready

 How to do it...

 How it works...

 Creating your first Rust program

 Getting ready

 How to do it...

How it works...

Defining a variable assignment

Getting ready

How to do it...

How it works...

Setting up Boolean and the character types

Getting ready

How to do it...

How it works...

Controlling decimal points, number formats, and named arguments

Getting ready

How to do it...

How it works...

Performing arithmetic operations

Getting ready

How to do it...

How it works...

Defining mutable variables

Getting ready

How to do it...

How it works...

Declaring and performing string operations

Getting ready

How to do it...

How it works...

Declaring arrays and using slices in Rust

Getting ready

How to do it...

How it works...

Declaring vectors in Rust

Getting ready

How to do it...

How it works...

Declaring tuples in Rust

Getting ready

How to do it...

How it works...

Performing calculations on two numbers

Getting ready

How to do it...

How it works...

2. Advanced Programming with Rust

Introduction

Defining an expression

Getting ready

How to do it...

How it works...

Defining constants

Getting ready

How to do it...

How it works...

Performing variable bindings

Getting ready

How to do it...

How it works...

Performing type casting in Rust

Getting ready

How to do it...

How it works...

Decision-making with Rust

Getting ready

How to do it...

How it works...

Looping operations in Rust

Getting ready

How to do it...

How it works...

Defining the enum type

Getting ready

How to do it...

How it works...

Defining closures

Getting ready

How to do it...

How it works...

Performing pointer operations in Rust

Getting ready

How to do it...

How it works...

Defining your first user-defined data type

Getting ready

How to do it...

How it works...

Adding functionality to the user-defined data type

Getting ready

How to do it...

How it works...

Similar functionality for different data type

Getting ready

How to do it...

How it works...

3. Deep Diving into Cargo

Introduction

Creating a new project using Cargo

Getting ready

How to do it...

How it works...

Downloading an external crate from crates.io

Getting ready

How to do it...

How it works...

Working on existing Cargo projects

Getting ready

How to do it...

How it works...

Running tests with Cargo

Getting ready

How to do it...

How it works...

Configuration management of the project

Getting ready

How to do it...

How it works...

Building the project on the Travis CI

Getting ready

How to do it...

How it works...

Uploading to crates.io

Getting ready

How to do it...

How it works...

4. Creating Crates and Modules

Introduction

Defining a module in Rust

Getting ready

How to do it...

How it works...

Building a nested module

Getting ready

How to do it...

How it works...

Creating a module with struct

Getting ready

How to do it...

How it works...

Controlling modules

Getting ready

How to do it...

How it works...

Accessing modules

Getting ready

How to do it...

How it works...

Creating a file hierarchy

Getting ready

How to do it...

How it works...

Building libraries in Rust

Getting ready

How to do it...

How it works...

Calling external crates

- Getting ready
- How to do it...
- How it works...

5. Deep Dive into Parallelism

- Introduction
- Creating a thread in Rust
 - Getting ready
 - How to do it...
 - How it works...

- Spawning multiple threads
 - Getting ready
 - How to do it...
 - How it works...

- Holding threads in a vector
 - Getting ready
 - How to do it...
 - How it works...

- Sharing data between threads using channels
 - Getting ready
 - How to do it...
 - How it works...

- Implementing safe mutable access
 - Getting ready
 - How to do it...
 - How it works...

- Creating child processes
 - Getting ready
 - How to do it...
 - How it works...

- Waiting for a child process
 - Getting ready
 - How to do it...
 - How it works...

- Making sequential code parallel
 - Getting ready
 - How to do it...
 - How it works...

6. Efficient Error Handling

- Introduction
- Implementing panic
 - Getting ready
 - How to do it...
 - How it works...
- Implementing Option
 - Getting ready
 - How to do it...
 - How it works...
- Creating map combinator
 - Getting ready
 - How to do it...
 - How it works...
- Creating and_then combinator
 - Getting ready
 - How to do it...
 - How it works...
- Creating map for the Result type
 - Getting ready
 - How to do it...
 - How it works...
- Implementing aliases
 - Getting ready
 - How to do it...
 - How it works...
- Handling multiple errors
 - Getting ready
 - How to do it...
 - How it works...
- Implementing early returns
 - Getting ready
 - How to do it...
 - How it works...
- Implementing the try! macro
 - Getting ready
 - How to do it...
 - How it works...
- Defining your own error types

- Getting ready
- How to do it...
- How it works...
- Implementing the boxing of errors
 - Getting ready
 - How to do it...
 - How it works...

7. Hacking Macros

- Introduction
- Building macros in Rust
 - Getting ready
 - How to do it...
 - How it works...
- Implementing matching in macros
 - Getting ready
 - How to do it...
 - How it works...
- Playing with common Rust macros
 - Getting ready
 - How to do it...
 - How it works...
- Implementing designators
 - Getting ready
 - How to do it...
 - How it works...
- Overloading macros
 - Getting ready
 - How to do it...
 - How it works...
- Implementing repeat
 - Getting ready
 - How to do it...
 - How it works...
- Implementing DRY
 - Getting ready
 - How to do it...
 - How it works...

8. Integrating Rust with Other Languages

Introduction
Calling C operations from Rust
 Getting ready
 How to do it...
 How it works...
Calling Rust commands from C
 Getting ready
 How to do it...
 How it works...
Calling Rust operations from Node.js apps
 Getting ready
 How to do it...
 How it works...
Calling Rust operations from Python
 Getting ready
 How to do it...
 How it works...
Writing a Python module in Rust
 Getting ready
 How to do it...
 How it works...

9. Web Development with Rust

Introduction
Setting up a web server
 Getting ready
 How to do it...
 How it works...
Creating endpoints
 Getting ready
 How to do it...
 How it works...
Handling JSONRequests
 Getting ready
 How to do it...
 How it works...
Building custom error handlers
 Getting ready
 How to do it...

How it works...
 Hosting templates

 Getting ready
 How to do it...
 How it works...

10. Advanced Web Development in Rust

 Introduction
 Setting up the API
 Getting ready
 How to do it...
 How it works...
 Saving user data in MongoDB
 Getting ready
 How to do it...
 Fetching user data
 Getting ready
 How to do it...
 How it works...
 Deleting user data
 Getting ready
 How to do it...
 How it works...

11. Advanced Rust Tools and Libraries

 Introduction
 Setting up rustup
 Getting ready
 How to do it...
 How it works...
 Setting up rustfmt
 Getting ready
 How to do it...
 How it works...
 Setting up rust-clippy
 Getting ready
 How to do it...
 How it works...
 Setting up and testing with Servo
 Getting ready

How to do it...

How it works...

Generating random numbers

Getting ready

How to do it...

How it works...

Writing lines to a file

Getting ready

How to do it...

How it works...

Parsing unstructured JSON

Getting ready

How to do it...

How it works...

Parsing URL from a string

Getting ready

How to do it...

How it works...

Decompressing a tarball

Getting ready

How to do it...

How it works...

Compressing a directory to a tarball

Getting ready

How to do it...

How it works...

Finding file extensions recursively

Getting ready

How to do it...

How it works...

Preface

Rust is a system programming language that runs blazingly fast, prevents segfaults, and guarantees thread safety. If you are building concurrent applications, server-side programs, or high-performance applications, you will benefit from this language. This book comes with a lot of application-specific recipes to kick-start your developing real-world, high-performance applications with the Rust programming language and integrating Rust units into your existing applications. In this book, you will find over 85 practical recipes written in Rust that will allow you to use the code samples in your existing applications right away.

This book will help you understand the core concepts of the Rust language, enabling you to develop efficient and high-performance applications by incorporating features such as zero cost abstraction and better memory management. We'll delve into advanced-level concepts such as error handling, macros, crates, and parallelism in Rust. Toward the end of the book, you will learn how to create HTTP servers and web services, building a strong foundational knowledge in server-side programming and enabling you to deliver solutions to build high-performance and safer production-level web applications and services using Rust.

What this book covers

[Chapter 1](#), *Let us Make System Programming Great Again*, provides a brief overview of how to get started with Rust programming by setting up the compiler and learning about various assignment operations and data types in Rust.

[Chapter 2](#), *Advanced Programming with Rust*, covers recipes that help in implementing expressions that will represent the state of the code, build logic using decision-making statements, and declare custom complex data type to represent a real-world scenario.

[Chapter 3](#), *Deep Diving into Cargo*, demonstrates the powerful features of the Cargo tool that is the Rust's package manager. This chapter contains recipes that help in creating, developing, packaging, maintaining, testing, and deploying Rust applications using the Cargo tool.

[Chapter 4](#), *Creating Crates and Modules*, shows you how to develop a highly modular production-grade Rust application. This chapter contains recipes that help in building libraries in Rust and also define control and access for features through external programs.

[Chapter 5](#), *Deep Dive into Parallelism*, contains recipes that will help you perform parallel operations to build a high-performance Rust application by learning the concurrency and parallelism features in Rust.

[Chapter 6](#), *Efficient Error Handling*, explores various recipes by which the developer can prepare to efficiently manage and handle errors in the Rust application.

[Chapter 7](#), *Hacking Macros*, teaches you how to create a Macro in the Rust programming language for creating powerful operations to execute specific tasks.

[Chapter 8](#), *Integrating Rust with Other Languages*, covers the techniques that will help us to create Rust units in our existing applications written in other

languages, such as C, Node.js, and Python.

[Chapter 9](#), *Web Development with Rust*, outlines recipes that will help the developer to learn and use the existing web framework libraries in Rust to set up a web server, handle web requests, and more.

[Chapter 10](#), *Advanced Web Development in Rust*, teaches you to build an end-to-end RESTful API solution using the Nickel crate in Rust language, which connects to MongoDB service and performs the GET, POST, and DELETE requests on user data from an endpoint.

[Chapter 11](#), *Advanced Rust Tools and libraries*, lists recipes that help us to set up various advanced Rust tools that help the developer to write production-level Rust code, catch errors, and deep dive into ground-level libraries in Rust, which performs common operations and functionalities out of the box.

What you need for this book

You will need the following software to complete all the recipes in this book:

- System running either Windows or Linux OS distribution with the minimum following resources:
 - Hard disk capacity: 2 GB
 - Processor: 1 GHz or higher (must support hyper-threading, multi-core CPUs)
 - Memory capacity: 2 GB or more recommended
- These recipes have been tested with stable Rust compiler versions of 1.14.0 and above
- We will need decent internet connection for downloading open source Rust libraries and other dependencies throughout the different recipes of the book

Who this book is for

The book is for developers, students, consultants, architects, and enthusiastic programmers across any vertical to learn about the state-of-the-art system programming language Rust and its unique features, which helps in delivering safe and high-performance production-level applications.

The book comes with a lot of application specific recipes, which will help the developers get kick started with developing web applications and high-performance Rust modules into their existing applications.

Sections

In this book, you will find several headings that appear frequently (Getting ready, How to do it..., How it works..., There's more..., and See also). To give clear instructions on how to complete a recipe, we use these sections as follows:

Getting ready

This section tells you what to expect in the recipe, and describes how to set up any software or any preliminary settings required for the recipe.

How to do it...

This section contains the steps required to follow the recipe.

How it works...

This section usually consists of a detailed explanation of what happened in the previous section.

There's more...

This section consists of additional information about the recipe in order to make the reader more knowledgeable about the recipe.

See also

This section provides helpful links to other useful information for the recipe.

Conventions

In this book, you will find a number of text styles that distinguish between different kinds of information. Here are some examples of these styles and an explanation of their meaning. Code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles are shown as follows: "A few other commands with the `rustup.sh` script are as follows."

A block of code is set as follows:

```
| fn main() {  
|     println!("Welcome to Rust Cookbook");  
| }
```

Any command-line input or output is written as follows:

```
| curl https://sh.rustup.rs -ssf | sh
```

New terms and important words are shown in bold.



In macOS, you can open the Terminal by pressing the F4 key, which opens the launchpad and searches for the Terminal.

Then, you can select the Terminal option in the display. This is just one possible way of doing it; there are other ways too.

In Linux distributions, we can jointly press Ctrl + Alt + T to open the Terminal or search for it in the application search window.



It's considered good style to put the opening curly brace on the same line as the function declaration, with one space in between.

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book-what you liked or disliked. Reader feedback is important for us as it helps us develop titles that you will really get the most out of. To send us general feedback, simply e-mail feedback@packtpub.com, and mention the book's title in the subject of your message. If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide at www.packtpub.com/authors.

Customer support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

Downloading the example code

You can download the example code files for this book from your account at <http://www.packtpub.com>. If you purchased this book elsewhere, you can visit <http://www.packtpub.com/support> and register to have the files e-mailed directly to you. You can download the code files by following these steps:

1. Log in or register to our website using your e-mail address and password.
2. Hover the mouse pointer on the SUPPORT tab at the top.
3. Click on Code Downloads & Errata.
4. Enter the name of the book in the Search box.
5. Select the book for which you're looking to download the code files.
6. Choose from the drop-down menu where you purchased this book from.
7. Click on Code Download.

You can also download the code files by clicking on the Code Files button on the book's webpage at the Packt Publishing website. This page can be accessed by entering the book's name in the Search box. Please note that you need to be logged in to your Packt account. Once the file is downloaded, please make sure that you unzip or extract the folder using the latest version of:

- WinRAR / 7-Zip for Windows
- Zipg / iZip / UnRarX for Mac
- 7-Zip / PeaZip for Linux

The code bundle for the book is also hosted on GitHub at <https://github.com/Packt Publishing/Rust-Cookbook>. We also have other code bundles from our rich catalog of books and videos available at <https://github.com/PacktPublishing/>. Check them out!

Downloading the color images of this book

We also provide you with a PDF file that has color images of the screenshots/diagrams used in this book. The color images will help you better understand the changes in the output. You can download this file from https://www.packtpub.com/sites/default/files/downloads/RustCookbook_ColorImages.pdf.

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books-maybe a mistake in the text or the code-we would be grateful if you could report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting <http://www.packtpub.com/submit-errata>, selecting your book, clicking on the Errata Submission Form link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded to our website or added to any list of existing errata under the Errata section of that title. To view the previously submitted errata, go to <https://www.packtpub.com/books/content/support> and enter the name of the book in the search field. The required information will appear under the Errata section.

Piracy

Piracy of copyrighted material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works in any form on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy. Please contact us at copyright@packtpub.com with a link to the suspected pirated material. We appreciate your help in protecting our authors and our ability to bring you valuable content.

Questions

If you have a problem with any aspect of this book, you can contact us at questions@packtpub.com, and we will do our best to address the problem.

Let us Make System Programming Great Again

In this chapter, we will cover the following recipes:

- Setting up Rust in Linux and macOS
- Setting up Rust in Windows
- Creating your first Rust program
- Defining a variable assignment
- Setting up Boolean and the character types
- Controlling decimal points, number formats, and named arguments
- Performing arithmetic operations
- Defining mutable variables
- Declaring and performing string operations
- Declaring arrays and using slices in Rust
- Declaring vectors in Rust
- Declaring tuples in Rust
- Performing calculations on two numbers

Introduction

This chapter is focused on bringing about a behavioral change in you in regard to setting up Rust and programming with it. The objective of this chapter is to make you realize why one should use Rust when there are so many other programming languages out there solving various problems in different verticals--why is there a need for yet another programming language?

These are the fundamental questions that would come to one's mind when they are learning a new language, well Rust is a systems programming language that runs blazingly fast, prevents segfaults, and guarantees thread safety. As the definition mentions Rust is focused towards eliminating a various class of system programming bugs and errors which at the end of the day helps in making secure, faster, and out-of-the-box production grade applications.

This chapter dives deeper into various assignment operations and their features, data types, and data structures in Rust.

Setting up Rust in Linux and macOS

We will explore ways in which we can install Rust components in Linux and macOS operating systems and also cover the different problems faced during the installation.

Getting ready

In order to run Rust code in your workstations, we have to install the Rust compiler. We require Unix-based operating systems such as Linux distributions and macOS.

How to do it...

Follow these steps to set up Rust on your system:

1. Open the Terminal.

In macOS, you can open the Terminal by pressing the F4 key, which opens the launchpad and searches for the Terminal.



Then, you can select the Terminal option in the display. This is just one possible way of doing it; there are other ways too.

In Linux distributions, we can jointly press Ctrl + Alt + T to open the Terminal or search for it in the application search window.

2. Type the following command to install the Rust compiler and Cargo in your system:

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

You can also try using the following command:

```
| curl -sf https://static.rust-lang.org/rustup.sh | sudo sh
```

The preceding commands will start the installation, and as it proceeds, the script will require user input. Enter for the default installation, which is the standard way. After this step, different components will be downloaded. If the installation happens without any error, you should be able to see the following screenshot:

```
info: syncing channel updates for 'stable-x86_64-unknown-linux-gnu'
info: downloading component 'rustc'
  47.1 MiB / 47.1 MiB (100 %) 307.2 KiB/s ETA: 0 s
info: downloading component 'rust-std'
  56.8 MiB / 56.8 MiB (100 %) 240.0 KiB/s ETA: 0 s
info: downloading component 'cargo'
  4.0 MiB / 4.0 MiB (100 %) 214.2 KiB/s ETA: 0 s
info: installing component 'rustc'
info: installing component 'rust-std'
info: installing component 'cargo'
info: default toolchain set to 'stable'

stable installed - rustc 1.14.0 (e8a012324 2016-12-16)
stable installed - ru
Rust is installed now. Great!
Rust is installed now.
To get started you need Cargo's bin directory in your PATH environment
variable. Next time you log in this will be done automatically.
To configure your current shell run source $HOME/.cargo/env
```

Rust is installed now. Great!

Uninstalling Rust

Uninstalling Rust is as easy as installing it. Open the Terminal and type the following command:

```
| rustup self uninstall
```

```
viki@Vigneshwer:~$ rustup self uninstall
```

Thanks for hacking in Rust!

This will uninstall all Rust toolchains and data, and remove \$HOME/.cargo/bin from your PATH environment variable.

Continue? (y/N) y

```
info: removing rustup home
info: removing cargo home
info: removing rustup binaries
info: rustup is uninstalled
viki@Vigneshwer:~$
```

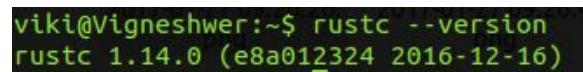
Troubleshooting

Rust's compiler version

If you have reached here, you have got Rust installed on your system, and you can go ahead and verify it. Open up the Terminal and enter the following command, which will give you the version of Rust installed:

```
| rustc --version
```

Take a look at the following screenshot:



```
viki@Vigneshwer:~$ rustc --version
rustc 1.14.0 (e8a012324 2016-12-16)
```

Here, `rustc` stands for the Rust compiler and `--version` displays the Rust version we have downloaded. By default, the `rustup.sh` script downloads the latest stable build. In this book, we are working with the `1.14.0` version.

Congrats, if you have reached this step without any error! Rust has been installed successfully.

Advanced installation options

A few other commands with the `rustup.sh` script are as follows. These commands are not necessary for common usage. The following commands are advanced commands that are not usually used by developers at an early stage of Rust development.

1. Install to a particular location:

```
|   rustup.sh prefix=my/install/dir
```

2. Save the download for faster reinstallation:

```
|   rustup.sh save
```

3. Install nightly:

```
|   rustup.sh channel=nightly
```

4. Install nightly archives:

```
|   rustup.sh --channel=nightly date=2015-04-09
```

5. Install the explicit versions:

```
|   rustup.sh --revision=1.0.0-beta
```

These commands help with the installation of a particular build, unstable releases, and version-controlled component installation.

Troubleshooting

If you try to reinstall Rust after its uninstallation, you'll often get an error saying that `rustup.sh` already exists. Please refer the following screenshot:

```
viki@Vigneshwer:~$ curl https://sh.rustup.rs -sSf | sh
info: downloading installer
warning: it looks like you have existing rustup.sh metadata
warning: rustup cannot be installed while rustup.sh metadata exists
warning: delete `/home/viki/.rustup` to remove rustup.sh
error: cannot install while rustup.sh is installed
rustup: command failed: /tmp/tmp.3rdcWllw1t/rustup-init
```

To solve the error, just delete the `.rustup` executable file from the user space:

```
| rm -rf /home/viki/.rustup
```

If the version command doesn't work for you, then you probably have the PATH environment variable wrong and have not included Cargo's binary directory, which is `~/.cargo/bin` on Unix and `%USERPROFILE%\.cargo\bin` on Windows.

This is the directory where Rust development tools are present, and most Rust developers keep it in their PATH environment variable, which makes it possible for them to run `rustc` on the command line.

Due to the differences between operating systems, command shells, and bugs in the installation, you may need to restart your shell, log out of the system, or configure PATH manually as appropriate to your operating environment.

Rust does not do its own linking, so you'll need to have a linker installed. Doing so will depend on your specific system. For Linux-based systems, Rust will attempt to call `cc` for linking. On Rust built on Windows with Microsoft Visual Studio, this depends on having Microsoft Visual C++ Build Tools installed. These do not need to be in `%PATH%`, as `rustc` will find them automatically. In general, if you have your linker in



a non-traditional location, you can call `rustc linker=/path/to/cc`, where `/path/to/cc` should point to your linker path.

If you are still stuck, there are a number of places where you can get help. The easiest is the **#rust--beginners** IRC channel for general discussion, and the **#rust** IRC channel, which we can access through Mibbit. Other great resources include the Rust user's forum and Stack Overflow.

How it works...

The shell script `rustup.sh` is a great way to install Rust and has been used by many developers to not only install Rust, but also Cargo on their machines.

The working of this script is pretty straightforward, where the code of the bash script hosted on the rustup server is downloaded on the host system and run automatically by passing the script code to the pipe symbol. The script offers you various installation options through which you can choose the version and type of Rust compiler you want to install. We have the nightly version, which is not the stable one, in Rust's nightly version. This version is used by developers and contributors to test and develop features for their existing projects.

Setting up Rust in Windows

This recipe covers how to set up Rust on a Windows system.

Getting ready

We will require a Windows machine for this purpose.

How to do it...

It is very easy to install it on Windows:

1. Download and run the `.exe` file from <https://win.rustup.rs>.
2. Click on the downloaded executable; this will start the installation in a Command Prompt.
3. Select option 1 in the Command Prompt for regular installation, which is recommended.

How it works...

It's similar to Linux and macOS; we have to download the executable and run it, which pops up the Command Prompt where the installation starts. Here, instead of using the shell script, the Rust team provides an executable file for Windows. The executable downloads the Rust compiler and Cargo dependencies on your host system.

Creating your first Rust program

This recipe is to help you make sure that your compiler is working right and also create a workspace where you can try out all these recipes.

Getting ready

We will require the Rust compiler setup on the host system for programming; I suggest you download a text editor of your choice for this. In this book, we are using the Sublime Text editor for the code development process.

How to do it...

1. Create a folder in your user space where you will be storing all the programs of the book:

```
| mkdir /home/viki/rust_cookbook
```

2. This command will create a directory for you in your user space:

```
| cd /home/viki/rust_cookbook
```

The preceding commands will take us to the particular directory.

3. Now, make a file named `sample.rs`; the `.rs` extension indicates that it is a Rust script.
4. Open the script in your text editor and enter the following code:

```
| fn main() {  
|     println!("Welcome to Rust Cookbook");  
| }
```

5. Save the file and go to your Terminal.
6. Compile the code with the Rust compiler, which will create the executable and run in a system without Rust on it:

```
viki@Vigneshwer:~/rust_cookbook$ rustc ./sample.rs  
viki@Vigneshwer:~/rust_cookbook$ ls  
sample sample.rs  
viki@Vigneshwer:~/rust_cookbook$ ./sample  
Welcome to Rust Cookbook
```


How it works...

Let's go through the code in brief and understand how it is being executed. The Rust compiler looks for the `main` function, which is the starting part of the code flow. We have a print statement and the dialogue to be displayed. When we compile the code, an executable is created which on execution will print the dialogue.

Rust files always end in a `.rs` extension. If you have to use more than one word in your filename to represent your project it is recommended to use an underscore to separate them for example, we would use `my_program.rs` rather than `myprogram.rs`, this helps in improving readability.

The keyword `fn` defines a function in Rust. The `main` function is the beginning of every Rust program. In this program, we have a `main` function which does not take any input arguments and returns any type. In case of any arguments, it would go in the parentheses `()`. The function body is wrapped in curly braces `{ }`. Rust requires these around all function bodies.



It's considered good style to put the opening curly brace on the same line as the function declaration, with one space in between.

The `println!` macro in the `std` Rust crate is used for printing to the standard output, with a newline. Which is inside the body of the `main` function and prints the string `Welcome to Rust Cookbook`. The line ends with a semicolon `(;)`. Rust is an expression-oriented language, which means that most things are expressions, rather than statements. The semicolon `(;)` indicates that this expression is over, and the next one is ready to begin.

Before running a Rust program, we have to compile it post which we can use an executable file to run the program and print the string in the terminal. The Rust compiler by entering the `rustc` command and passing it the name

of your source file will create the executable, for example, `rustc main.rs` would create an executable `./main`. in the same directory.

Defining a variable assignment

We will dive deeply into the various assignment operators and functions in this section.

Getting ready

We would require the Rust compiler and any text editor for coding and create a file named `assignment.rs` in the project workspace.

How to do it...

1. Start by declaring the different libraries we would be using for this recipe in the `assignment.rs` script:

```
// Primitive libraries in rust
use std::{i8,i16,i32,i64,u8,u16,u32,u64,f32,f64,isize,usize};
use std::io::stdin;
```

The `use` statement tells the compiler that the program would use the following properties of the library. The `std` is an inbuilt library that comes along with the Rust compiler and doesn't need to be externally downloaded. `i8` and `i16` are different data types of the variable that will be used in the program, and `stdin` helps us accept user input from the user:

```
fn main() {
    println!("Understanding assignment");
    // Compiler will automatically figure out the data type if
    not mentioned
    // Cannot change the value
    let num =10;
    println!("Num is {}", num);
}
```

2. The output of the preceding script is as follows:

```
viki@Vigneshwer:~/rust_cookbook$ rustc -A warnings ./assignment.rs
viki@Vigneshwer:~/rust_cookbook$ ls
assignment assignment.rs sample sample.rs
viki@Vigneshwer:~/rust_cookbook$ ./assignment
Understanding assignment
Num is 10 main()
```

3. Replace the `main` function of the preceding script in `assignment.rs` file with the following code snippet below:

```
fn main(){
    let age: i32 =40;
    println!("Age is {}", age);
    // Prints the max and min value of 32bit integer
    println!("Max i32 {}",i32::MAX);
    println!("Min i32 {}",i32::MIN);
}
```

```
viki@Vigneshwer:~/rust_cookbook$ ./assignment
Age is 40
Max i32 2147483647
Max i32 -2147483648
viki@Vigneshwer:~/rust_cookbook$
```

4. In the previous code snippet, we declared a variable named `age` and explicitly told the compiler that it's a 32-bit integer type data and that we are printing the maximum and minimum values of the 32-bit int data type. Now, the next piece of code:

```
// Another way of variable assigning
let(f_name,l_name)=("viki","d");
    println!("First name {0} and last name {1}",f_name,l_name);
```

Here, we have declared two variables in `f_name` and `l_name` using brackets(). This is a way to declare multiple variables in a single statement in Rust. Similarly, while printing them, we can number the position of the variable to determine which variable has to be printed first.

How it works...

In the previous section, we declared the `main` function from where the execution would start. Here, we are declaring a variable named `num` and assigning it to a value of 10. The `let` statement enables you to declare variables and assigns them to a value. We are not explicitly telling what kind of data type the `num` variable is, but during compilation time, it will be automatically determined and memory will be assigned based on that.

The `num` value is immutable, which means that its value cannot be changed during the program, and it will be removed from the memory once it goes out of the scope of the `main` function. To print the value of the number, we have to use braces; we will cover more of this in detail in the next section.

Setting up Boolean and the character types

Boolean operators are of great help to programmers for state identification and checking. In this recipe, you will learn about the assignment of character type variables.

Getting ready

We would require the Rust compiler and any text editor for coding.

How to do it...

1. Create a file named `boolean.rs` and compile the following code:

```
fn main() {  
    //Setting boolean and character types  
    let bool_val: bool = true;  
    let x_char: char = 'a';  
  
    // Printing the character  
    println!("x char is {}", x_char);  
    println!("Bool value is {}", bool_val);  
}
```

2. In the preceding code snippet, we are assigning a Boolean type variable and character values in Rust.
3. The output is as follows:

```
viki@Vigneshwer:~/rust_cookbook/chapter1_code$ rustc -A warnings ./boolean.rs  
viki@Vigneshwer:~/rust_cookbook/chapter1_code$ ./boolean  
x char is a  
Bool value is true
```


How it works...

In the preceding code snippet, we declared the `main` function where we defined two variables: `bool_val` and `x_char`. We assigned them with a Boolean and character value using the `let` statement. We followed this up by printing them.

Controlling decimal points, number formats, and named arguments

This recipe focuses on how to manipulate the print macros to perform various control operations in the data.

Getting ready

We would require the Rust compiler and any text editor for coding.

How to do it...

1. Enter the following code in a Rust script named `decimal.rs` and compile them:

```
fn main(){

    // Prints the first 2 numbers after the decimal points
    println!(" {:.2}", 1.2345 );
    println!("=====");

    // print the binary hex and octal format
    println!("B: {:b} H: {:x} O: {:o}", 10, 10, 10 );
    println!("=====");

    // Shifts
    println!("{}{:>ws$}", ten=10, ws=5 );
    println!("{}{:>0ws$}", ten=10, ws=5 );
}
```

2. The output of the code is as follows:

```
vikki@Vigneshwer:~/rust_cookbook$ rustc -A warnings ./decimal.rs
vikki@Vigneshwer:~/rust_cookbook$ ./decimal
1.23
=====
B: 1010 H: a O: 12
=====
    10
00010
```


How it works...

In the first print statement, we controlled the number of decimal points to be displayed in the Terminal. In the preceding code snippet, we set the value to be two after the colon symbol (:) in the print statement, which tells the compiler to only print the first two decimal points of the variable in the runtime.

The next print statement displayed Rust's built-in feature that can convert the value to be printed in a different number format. We printed the binary, hex, and octal value of the decimal value 10. To perform this activity, we specifically mentioned the parameter after the colon symbol in the print statement. This is understood by the Rust compiler. At runtime, Rust would automatically convert the decimal type into the mentioned number format, where `b` stands for binary, `x` for hex, and `o` for octal. This has to be given after `:` in the print statement.

Next, the `print` statement named arguments and we defined the **white space (ws)** type we wanted to. We have two arguments here: `ten` and `ws`. We had control over how we wanted to print the data and what kind of values we wanted to fill `ws` with. In the first print statement, we filled it with blank spaces. In the second print statement, we explicitly mentioned zero, which is what we want to fill the gaps with. We declared the named argument inside the curly braces of the print statement and assigned its data value.

Performing arithmetic operations

This recipe showcases the different types of arithmetic operations in Rust.

Getting ready

We would require the Rust compiler and any text editor for coding.

How to do it...

1. Create a file named `arithmetic.rs` in the workspace and compile the following code:

```
fn main(){
    // Arithmetic Operations
    println!("5 + 4 = {}", 5+4 );
    println!("5 - 4 = {}", 5-4 );
    println!("5 * 4 = {}", 5*4 );
    println!("5 / 4 = {}", 5/4 );
    println!("5 % 4 = {}", 5%4 );

    // Assigning data types and mathematical Operations
    let neg_4 = -4i32;
    println!("abs(-4) = {}", neg_4.abs() );
    println!("abs(-4) = {}", neg_4.pow(2) );
    println!("round(1.2345) = {}", 1.2354f64.round() );
    println!("ceil(1.2345) = {}", 1.2345f64.ceil() );
    print!("sin 3.14 = {}", 3.14f64.sin() );
}
```

2. We would get the following output:

```
viki@Vigneshwer:~/rust_cookbook$ rustc -A warnings ./arithmetic.rs
viki@Vigneshwer:~/rust_cookbook$ ./arithmetic
5 + 4 = 9
5 - 4 = 1
5 * 4 = 20
*****println!("*****");
5 / 4 = 1 // Assigning data types and mathematical Operations
5 % 4 = 1
*****let neg_4 = -4i32;
*****println!("abs(-4) = {}", neg_4.abs() );
abs(-4) = 4
abs(-4) = 16
round(1.2345)=1
ceil(1.2345)=2
sin 3.14 = 0.0015926529164868282
viki@Vigneshwer:~/rust_cookbook$
```


How it works...

In the first set of print statements, we have different types of arithmetic operations being performed on the data set during runtime. The following symbols in the brackets associated with each operation are used to perform the arithmetic operation:

- addition (+)
- subtraction (-)
- multiplication (x)
- division (/)
- modulus (%)

In the next set of print statements, we performed various mathematical operations, which come built in with the Rust compiler. We declared a variable named `neg_4` and assigned it the value `4i32`, which is a negative 32-bit integer with the value `4`. We set the absolute value of the variable by calling the `abs()` function with the variable name `variable.function`. Similarly, we performed other mathematical operations, such as `pow(value)`, which calculates and applies the power value to the data. The `round()` function finds the data to the nearest lower value. The `ceil()` function returns the smallest integer that is greater than or equal to the number. And, the `sin()` functions return the sine value.

Defining mutable variables

Rust has the unique feature of ownership and borrowing that enables it to overcome segfaults and data races. This is achieved by the mutability property of Rust.

Getting ready

We would require the Rust compiler and any text editor for coding.

How to do it...

1. Create a file named `mutable.rs` and compile the following code:

```
fn main() {
    let mut sample_var = 10;
    println!("Value of the sample variable is {}", sample_var);
    let sample_var = 20;
    println!("New Value of the sample variable is {}", sample_var);
}
```

2. You should get the following output:

```
viki@Vigneshwer:~/rust_cookbook$ rustc -A warnings ./mutable.rs
viki@Vigneshwer:~/rust_cookbook$ ./mutable
Value of the sample variable is 10
New Value of the sample variable is 20
```


How it works...

Since we have declared the variable type mutable, the Rust compiler allows the developer to change the data value assigned any number of times in the scope of the functions.

In the preceding program, we created a variable named `sample_var` and explicitly marked it as mutable type during its assignment. Due to this action, the Rust compiler allows the variables to be assigned different values.

Declaring and performing string operations

This recipe dives deeply into various built-in string operations and functions that let the developer split and modify string data.

Getting ready

We would require the Rust compiler and any text editor for coding.

How to do it...

1. Create a script named `string.rs` and compile the following code:

```
fn main() {
    // declaring a random string
    let rand_string = "I love Rust cookbook <3";

    // printing the length of the string
    println!("length of the string is {}", rand_string.len());
}
```

2. Here, you are creating an immutable variable called `rand_string` and assigning it with a string value. You are also using the `len()` function of the string data type to print the length of the sentence, which would also count the white space:

```
// Splits in string
let (first,second) = rand_string.split_at(7);
println!("First part : {0} Second part : {1}",first,second);
```

3. The `split_at(value)` function divides the sentence into two parts and assigns them as two variables: `first` and `second`. We then print them:

```
// Count using iterator count
let count = rand_string.chars().count();
print!("count {}",count);
```

4. We are using two functions here, namely `chars` and `count`, where `chars` identifies all the characters and `count` gives the aggregated number of characters:

```
println!("_____");
// printing all chars
let mut chars = rand_string.chars();
let mut indiv_chars = chars.next();
loop {
    // Its like switch in c++
    match indiv_chars {
        Some(x) => println!("{}" ,x),
        None => break
    }
    indiv_chars = chars.next();
}
```

5. In the preceding piece of code, we created a variable named `chars`, which has all the characters of the sentence. Then, in the next step, we created another variable named `indiv_chars`, which contains the first character of the `chars` variable. We used the `next()` functions to assign the value to `indiv_chars`.
6. In the `loop` function, print all the values of the `chars` variable until it becomes null:

```
    println!("_____");
    // iterate over whitespace
    let mut iter = rand_string.split_whitespace();
    let mut indiv_word = iter.next();
    loop {
        // Its like switch in c++
        match indiv_word {
            Some(x) => println!("{}" ,x),
            None => break
        }
        indiv_word = iter.next();
    }
```

7. In this section of the code, we will iterate over the using the built-in `split_whitespace()` function. This section would print the complete words:

```
    println!("_____");
    // iterate over next line
    let rand_string2 = "I love \n everything about \n Rust <3";
    let mut iter_line = rand_string2.lines();
    let mut indiv_sent = iter_line.next();
    loop {
        // Its like switch in c++
        match indiv_sent {
            Some(x) => println!("{}" ,x),
            None => break
        }
        indiv_sent = iter_line.next();
    }
```

8. In this section of the code, we will iterate over the next line using the built-in `lines()` function. This section would print the complete sentence.
9. You should get the following result:

```
viki@Vigneshwer:~/rust_cookbook/chapter1_code$ ./string
length of the string is 23
First part : I love  Second part : Rust cookbook <3
count 23
I
l
o
v
e
R
u
s
t
c
o
o
k
b
o
o
k
<
3
-----
I
love
Rust
cookbook
<3
-----
I love
    everything about
    Rust <3
viki@Vigneshwer:~/rust_cookbook/chapter1_code$ █
```


How it works...

In this recipe, we assign a variable named `rand_string` with a string value of `I love Rust cookbook` in which we perform a certain set of string operations.

In the first print statement, we display the length of the string by the `len()` method of the `str` type by calling `rand_string.len()` and correspondingly in the next print statement we use the `split_at()` method which expects an input argument which splits the string at the value of the argument passed and in our case we call `rand_string.split_at(7)` to split at index `7` of the string and assign it to the two variables named `first` and `second`, here space occupies an index value.

In the third print statement, we print the numbers of characters present by using the `chars()` and `count()` methods, we do it with the following syntax `rand_string.chars().count()` where `chars()` iterates through the characters and `count()` for counting the number of elements.

The `loop` is an iterative statement that keeps on running until the break key is called. We used the `match` function, which is similar to switch in other languages. We created different conditions for it to check and perform certain actions.

In the `match` statements, we have two conditions which are `Some(x)` and `None`. In both the cases, we perform unique operations which are as follows:

- In the case of `Some(x)`, it basically says that `x` is a value that exists and has a type `T` so we can use the `x` variable for further operations and in such cases, we print the value
- In the case of `None`, it refers to cases where a value does not exist which ideally is the end of the string and the operation we perform at these cases are to break the loop

We iterate over three types of conditions:

- The first set is individual cases, where we print all the characters of the string and is performed by the `chars()` and `next()` functions with the preceding `match` statements
- The second set is where we print all the different words of the string by splitting in the places of white spaces and is performed by the `split_whitespace()` and `next()` functions with the preceding `match` statements
- The last ones where we print all the different lines of the string by splitting in the places of next line and is performed by the `lines()` and `next()` functions with the preceding `match` statements

Declaring arrays and using slices in Rust

An array is a collection of objects that are of the same data type and is stored in contiguous memory. Arrays are always created using brackets `[]`, and their size is known at compile time. It is part of their type: `[T; size]`.

Slices are similar to arrays but their size is not known at compile time. The first mark is a pointer value to the data; the second mark is the length of the slice that is selected by the user depending on the application. Slices are usually used to borrow a section of an array and have the type signature and `[T]`.

Getting ready

We would require the Rust compiler and any text editor for coding.

How to do it...

1. Create a file named `array.rs` and compile the following code:

```
fn main() {  
    // Defining an array  
    let rand_array = [1,2,3];  
    println!("random array {:?}",rand_array );  
  
    // indexing starts with 0  
    println!("random array 1st element {}",rand_array[0] );  
    println!("random array length {}",rand_array.len() );  
  
    // last two elements  
    println!("random array {:?}",&rand_array[1..3] );  
}
```

2. We would get the following output:

```
viki@Vigneshwer:~/rust_cookbook$ rustc -A warnings ./array.rs  
viki@Vigneshwer:~/rust_cookbook$ ./array  
random array [1, 2, 3]  
random array 1st element 1  
random array length 3  
random array [2, 3]
```


How it works...

Here, we declared a variable named `rand_array` and assigned it to an array that has three elements inside square brackets: one, two, and three. In the first print statement, we had a question mark (?) after the colon (:), which indicates to the print statement that it would print all the elements of the array.

We can address each element of the array by the indices (which refer to the position of the array data element). In Rust, the positioning starts from zero. So when we print `rand_array[0]`, it will print the first element. In the third print statement, we used the `len()` function to get the length or the number of elements in the array; we used the length function by calling `rand_var.len()`.

In the fourth print, we had a new concept called slices. Arrays are borrowed as slices, where we mention the starting value of the pointer and length in the signature. Slices basically reference to a contiguous sequence of the element of the array instead of the whole element.

We printed the complete array by printing `&rand_var`, which would print the total array. We also borrowed a section of array using `&rand_var[1..3]`, where we mention the size of the slice in square brackets. Here, starting from one, we print all the sections of the numbers until the upper limit of three where one and three are the index values of the arrays which were dereferenced and printed in the Terminal.

Declaring vectors in Rust

Vector is a very important concept in programming and often people get confused between arrays and vectors. Let's demystify the concept. A vector is a dynamic array, which implies that its data allocation can be increased or decreased at runtime, and we will see how this can be done with a simple built-in function. Vectors are implemented in Rust as standard library types, namely `vec<T>`. Here, `T` means that we can have vectors of any data type; vectors always allocate their data on the heap and we can create them with the `vec!` macro.

Getting ready

We would require the Rust compiler and any text editor for coding.

How to do it...

1. Create a file named `vector.rs` and compile the following code:

```
| fn main() {  
|     // declaring a vector  
|     let mut vec1 = vec![1,2,3,4,5];
```

2. Declare a mutable vector `vec1` with five elements:

```
|     // printing element 3 in vector  
|     println!("Item 3 : {}", vec1[2]);
```

Here, to print the third element of the vector, we can refer to the particular data of the vector in the heap by its position.

3. The value of the position starts from zero and goes until $n-1$ if there are n data values:

```
|     // iterating in a vector  
|     for i in &vec1 {  
|         println!("{}: {}", i, *i)  
|     }
```

4. In the previous step, we iterated over the vector by taking reference of the vector and printing all its elements:

```
|     // push an element to vector  
|     vec1.push(6);  
|     println!("vector after push {:?}", vec1);  
  
|     // pop an element from vector  
|     vec1.pop();  
|     println!("vector after pop {:?}", vec1);
```

5. Add and remove the values from the vector.
6. You should get the following output:

```
vikiv@Vigneshwer:~/rust_cookbook$ rustc -A warnings ./vector.rs
vikiv@Vigneshwer:~/rust_cookbook$ ./vector
Item 3 : 3
1
2
3
4
5
vector after push [1, 2, 3, 4, 5, 6]
vector after pop [1, 2, 3, 4, 5]
```


How it works...

We cannot use the vector again if we have iterated by taking ownership of the vector, and for reiterating the vector many times, we have to take a reference to the vector.

Using `pop` and `push`, we can add/remove elements to/from the heap of the memory allocation, where the vector data is stored. This feature allows the vector to be dynamic. `push` adds a new element to the top, which is the last element of the indices, and `pop` removes the first value and last element with respect to the indices.

Declaring tuples in Rust

A tuple is a unique data structure and is widely used by many developers in their day-to-day development for processing values of different data types.

Generally, tuples are constructed by parentheses `()`, and each tuple has a type signature, such as `t1, t2, ...`, where `t1` and `t2` are the types of its member values.

Tuples are very handy in Rust when we want to return multiple data types. We can use Rust to package them into a single data structure.

Getting ready

We would require the Rust compiler and any text editor for coding.

How to do it...

1. Create a file named `tuple.rs` and compile the following code:

```
use std::i8;
fn main() {

    // Declaring a tuple
    let rand_tuple = ("Rust", 2017);
    let rand_tuple2 : (&str, i8) = ("Viki", 4);

    // tuple operations
    println!(" Name : {}", rand_tuple2.0);
    println!(" Lucky no : {}", rand_tuple2.1);
}
```

2. You should get the following output:

```
viki@Vigneshwer:~/rust_cookbook/chapter1_code$ rustc -A warnings tuples.rs
viki@Vigneshwer:~/rust_cookbook/chapter1_code$ ./tuples
Name : Viki
Lucky no : 4
viki@Vigneshwer:~/rust_cookbook/chapter1_code$ █
```


How it works...

In the previous section, we declared two tuples in two different ways. In the first method, the Rust compiler automatically located the data types. In the second method, we explicitly mentioned the data types and the tuples allowed us to create different data types.

In the tuple operation section, we extracted values from the tuple using tuple indexing, which is performed by printing `tuple_variable.index_value`.

Performing calculations on two numbers

This recipe covers all the aspects that we have learned from other recipes. We perform various operations, such as entering two values from the Terminal to be accepted as standard input by the user, converting the string that is acceptable to the integer, and performing arithmetic operations.

Getting ready

We would require the Rust compiler and any text editor for coding.

How to do it...

1. Create a file named `calculator.rs` and compile the following code:

```
// Libraries in rust
use std::io;
use std::{i32};
```

2. The `io` function helps us accept user input in the Terminal:

```
// Main Functions
fn main() {
    // Request for entering number 1
    println!("Enter First number ? ");
    let mut input1 = String::new();
    io::stdin().read_line(&mut input1).expect("Failed to read
line");

    // Request for entering number 2
    println!("Enter second number ? ");
    let mut input2 = String::new();
    io::stdin().read_line(&mut input2).expect("Failed to read
line");

    // Converting string to integer
    let aint: i32 = input1.trim().parse().ok().expect("Program
only
processes numbers, Enter number");
    let bint: i32 = input2.trim().parse().ok().expect("Program
only
processes numbers, Enter number");

    // Output of basic operations
    println!("sum is: {}", aint + bint);
    println!("difference is: {}", aint - bint);
    println!("Multiply is: {}", aint * bint);
    println!("division is: {}", aint / bint);
}
```

3. You should get the following output:

```
viki@Vigneshwer:~/rust_cookbook$ rustc -A warnings ./calculator.rs
viki@Vigneshwer:~/rust_cookbook$ ./calculator
Enter First number ? 
Enter second number ? 
20
Enter second number ? 
10
Enter second number ? 
10
sum is: 30
difference is: 10
multiply is: 200
division is: 2
viki@Vigneshwer:~/rust_cookbook$
```


How it works...

The `std::io` module contains a number of common things you'll need when. The core part of this module is the read and write traits.

We called the `read_line()` method from the `std::io` module. The `read_line()` method doesn't take a String as an argument; it takes a `&mut String`. Rust has a feature called *references*, which allows you to have multiple references to one piece of data, which can reduce copying. The job of `read_line` is to take what the user types as standard input and place it into a string. So it takes that string as an argument, and in order to add the input, it needs to be mutable. In this case, `io::Result` had an `expect()` method that took a value it was called on; if this isn't successful, our program will crash, displaying a message. If we do not call `expect()`, our program will compile but we'll get a warning.

For converting the string into an integer, we used `trim` and `parse` methods. The `trim()` method on the strings eliminates any `ws` at the beginning and end of the data. This means that if we type 5 and hit return, guess it would look like this: `5\n`. The `\n` represents a *newline*, the enter key induced newline will be got rid by `trim()` method, leaving behind our string with only the number 5.

The `parse()` method on string data parses a string into some kind of number format, since it can parse a variety of numbers, we need to give Rust a hint as to the exact type of number format we want it to convert, that is the reason why we have the statement `let a_int: i32`. The colon `(:)` after tells Rust we're going to annotate its type, `i32` is an integer containing 32-bit size. We used the `expect()` method to capture a crash in the cases of an error. The last section of the script was used to conduct regular arithmetic operations.

Advanced Programming with Rust

In this chapter, we will cover the following:

- Defining an expression
- Defining constants
- Performing variable bindings
- Performing type casting in Rust
- Decision-making with Rust
- Looping operations in Rust
- Defining the enum type
- Defining closures
- Performing pointer operations in Rust
- Defining your first user-defined data type
- Adding functionality to the user-defined data type
- Similar functionality for different data type

Introduction

This chapter is focused on equipping you with all the recipes to implement expressions that will represent the state of the code, build logic using decision-making statements such as `if...else`, declare a custom complex data type to represent a real-world scenario using `struct`, add functionality to a complex data type using traits, and control code execution using the looping statement.

Defining an expression

An expression, in simple words, is a statement in Rust by using which we can create logic and workflows in the program and applications. We will deep dive into understanding expressions and blocks in Rust.

Getting ready

We will require the Rust compiler and any text editor for coding.

How to do it...

Follow the ensuing steps:

1. Create a file named `expression.rs` with the next code snippet.
2. Declare the `main` function and create the variables `x_val`, `y_val`, and `z_val`:

```
// main point of execution
fn main() {

    // expression
    let x_val = 5u32;

    // y block
    let y_val = {
        let x_squared = x_val * x_val;
        let x_cube = x_squared * x_val;

        // This expression will be assigned to `y_val`
        x_cube + x_squared + x_val
    };

    // z block
    let z_val = {
        // The semicolon suppresses this expression and `()` is
        // assigned to `z`
        2 * x_val;
    };

    // printing the final outcomes
    println!("x is {:?}", x_val);
    println!("y is {:?}", y_val);
    println!("z is {:?}", z_val);
}
```

You should get the ensuing output upon running the code. Please refer to the following screenshot:

```
[vigneshwers-MacBook-Air:chapter2 vigneshwerdhinakaran$ rustc -A warnings expression.rs
[vigneshwers-MacBook-Air:chapter2 vigneshwerdhinakaran$ ./expression
x is 5
y is 155
z is ()
```


How it works...

All the statements that end in a semicolon (;) are expressions. A block is a statement that has a set of statements and variables inside the {} scope. The last statement of a block is the value that will be assigned to the variable. When we close the last statement with a semicolon, it returns () to the variable.

In the preceding recipe, the first statement which is a variable named `x_val`, is assigned to the value 5. Second, `y_val` is a block that performs certain operations on the variable `x_val` and a few more variables, which are `x_squared` and `x_cube` that contain the squared and cubic values of the variable `x_val`, respectively. The variables `x_squared` and `x_cube`, will be deleted soon after the scope of the block.

The block where we declare the `z_val` variable has a semicolon at the last statement which assigns it to the value of (), suppressing the expression. We print out all the values in the end.

We print all the declared variables values in the end.

Defining constants

Rust provides the ability to assign and maintain constant values across the code in Rust. These values are very useful when we want to maintain a global count, such as a timer threshold for example. Rust provides two `const` keywords to perform this activity. You will learn how to deliver constant values globally in this recipe.

Getting ready

We will require the Rust compiler and any text editor for coding.

How to do it...

Follow these steps:

1. Create a file named `constant.rs` with the next code snippet.

2. Declare the global `UPPERLIMIT` using `constant`:

```
// Global variables are declared outside scopes of other
function
const UPPERLIMIT: i32 = 12;
```

3. Create the `is_big` function by accepting a single integer as input:

```
// function to check if bunber
fn is_big(n: i32) -> bool {
    // Access constant in some function
    n > UPPERLIMIT
}
```

4. In the `main` function, call the `is_big` function and perform the decision-making statement:

```
fn main() {
    let random_number = 15;

    // Access constant in the main thread
    println!("The threshold is {}", UPPERLIMIT);
    println!("{} is {}", random_number, if
        is_big(random_number) { "big" } else { "small"
    });

    // Error! Cannot modify a `const`.
    // UPPERLIMIT = 5;

}
```

You should get the following screenshot as output upon running the preceding code:

```
[vigneshwers-MacBook-Air:chapter2 vigneshwerdhinakaran$ rustc -A warnings constant.rs
[vigneshwers-MacBook-Air:chapter2 vigneshwerdhinakaran$ ./constant
The threshold is 12
15 is big
```


How it works...

The workflow of the recipe is fairly simple, where we have a function to check whether an integer is greater than a fixed threshold or not. The `UPPERLIMIT` variable defines the fixed threshold for the function, which is a constant whose value will not change in the code and is accessible throughout the program.

We assigned `15` to `random_number` and passed it via `is_big (integer value)`; and we then get a boolean output, either `true` or `false`, as the return type of the function is a `bool` type. The answer to our situation is `false` as `15` is not bigger than `12`, which the `UPPERLIMIT` value set as the constant. We performed this condition checking using the `if...else` statement in Rust.

We cannot change the `UPPERLIMIT` value; when attempted, it will throw an error, which is commented in the code section.



Constants declare constant values. They represent a value, not a memory address: `type = value;`

Performing variable bindings

Variable binding refers to how a variable in the Rust code is bound to a type. We will cover pattern, mutability, scope, and shadow concepts in this recipe.

Getting ready

We will require the Rust compiler and any text editor for coding.

How to do it...

Perform the following step:

1. Create a file named `binding.rs` and enter a code snippet that includes declaring the `main` function and different variables:

```
fn main() {
    // Simplest variable binding
    let a = 5;
    // pattern
    let (b, c) = (1, 2);
    // type annotation
    let x_val: i32 = 5;
    // shadow example
    let y_val: i32 = 8;
    {
        println!("Value assigned when entering the
            scope : {}", y_val); // Prints "8".
        let y_val = 12;
        println!("Value modified within scope :{}", y_val);
        // Prints "12".
    }
    println!("Value which was assigned first : {}", y_val);
    // Prints "8".
    let y_val = 42;
    println!("New value assigned : {}", y_val);
    //Prints "42".
}
```

You should get the following screenshot as output upon running the preceding code:

```
[vigneshwers-MacBook-Air:chapter2 vigneshwerdhinakaran$ rustc -A warnings binding.rs
[vigneshwers-MacBook-Air:chapter2 vigneshwerdhinakaran$ ./binding
Value assigned when entering the scope : 8
Value modified within scope :12
Value which was assigned first : 8
New value assigned : 42
```


How it works...

The `let` statement is the simplest way to create a binding, where we bind a variable to a value, which is the case with variable `a`. To create a pattern with the `let` statement, we assign the pattern values to `b` and `c` values in the same pattern. Rust is a statically typed language. This means that we have to specify our types during an assignment, and at compile time, it is checked to see if it is compatible. Rust also has the type reference feature that identifies the variable type automatically at compile time. The `variable_name : type` is the format we use to explicitly mention the type in Rust. We read the assignment in the following format:

x_val is a binding with the type i32 and the value 5.

Here, we declared `x_val` as a 32-bit signed integer. However, Rust has many different primitive integer types that begin with `i` for signed integers and `u` for unsigned integers, and the possible integer sizes are 8, 16, 32, and 64 bits.

Variable bindings have a scope that makes the variable alive only in the scope. Once it goes out of the scope, the resources are freed.

A block is a collection of statements enclosed by `{}`. Function definitions are also blocks! We use a block to illustrate the feature in Rust that allows variable bindings to be shadowed. This means that a later variable binding can be done with the same name, which in our case is `y_val`. This goes through a series of value changes, as a new binding that is currently in scope overrides the previous binding. Shadowing enables us to rebinding a name to a value of a different type. This is the reason why we are able to assign new values to the immutable `y_val` variable in and out of the block.

Performing type casting in Rust

In this recipe, you will learn about casting between different data types in Rust. Rust does not provide an automatic type cast. The developer has to manually own it. Using as we will perform safe type casting in Rust.

Getting ready

We will require the Rust compiler and any text editor for coding.

How to do it...

Perform the following steps:

1. Create a file named `typecasting.rs` and enter the following code to the script:

```
use std::i32,f32;

// Sample function for assigning values to
confusion matrix
fn main() {

    // assigning random values to the confusion matrix
    let(true_positive,true_negative,false_positive,
    false_negative)=(100,50,10,5);

    // define a total closure
    let total = true_positive + true_negative +
    false_positive + false_negative;

    println!("The total predictions {}",total);

    // Calculating the accuracy of the model
    println!("Accuracy of the model
    {:.2}",percentage(accuracy(true_positive,
    true_negative,total)));
}
```

2. In the preceding code snippet, we created four variables: `true_positive`, `true_negative`, `false_positive`, and `false_negative`. These are basically the four measurement parameters of a confusion matrix.
3. Call the `accuracy` and `percentage` function that returns the final accuracy percentage.
4. The `total` variable is the sum of all the measurements:

```
// Accuracy Measures the overall performance of
the model
fn accuracy(tp:i32,tn:i32,total:i32) -> f32 {
    // if semi-colon is not put then that returns
    // No automatic type cast in rust
    (tp as f32 + tn as f32 )/(total as f32)
}

// Converting to percentage
fn percentage(value:f32) -> f32 {
    value *100.0
}
```

5. The `accuracy` function accepts those which are all `int` data types that return a `float` data type.
6. The value received from the `accuracy` function is passed to the `percentage` function and the `accuracy` is printed.

You should get the following screenshot as output upon running the preceding code:

```
[vigneshwers-MacBook-Air:chapter2 vigneshwerdhinakaran$ rustc -A warnings typecasting.rs
[vigneshwers-MacBook-Air:chapter2 vigneshwerdhinakaran$ ./typecasting
The total predictions 165
Accuracy of the model 90.91
```


How it works...

In this recipe, we have two functions, `accuracy` and `percentage`, which take in arguments from the `main` function and convert the type passed to the desired type, due to the nature of the arithmetic operations for which we use the `as` keywords in Rust which helps in type casting in Rust. In the case of the `accuracy` function, it takes three input arguments of type `i32` and returns a single `f32` type value.

In order to protect developers from accidental casts, Rust makes it mandatory for developers to convert data types manually. In the following example, we define an `int` variable named `a` and assign it the value `3`; after the assignment operation, we would see that a part of the code is commented. This implies that it won't be compiled by the Rust compiler. If we take a careful look at the code, we find that we are multiplying an `int` variable with a float value, which will give us a type mismatch error during compilation:

```
| let a = 3;
| /*
| let b = a * 0.2; //Won't compile
| */
```

As we can see, we used the `as` keyword converting `int` to `float` (64-bit) in order to multiply an `int` value by a `float` variable. This step produced `b` without any error:

```
| let b = a as f64 * 0.2;
```



Note that when we perform arithmetic operations in the same kind of data type, we don't have to worry about type conversion as the result of the operation produced is automatically typecasted.

Decision-making with Rust

In this recipe, we will learn about decision-making statements in Rust. Condition checking in Rust is similar to other dynamic programming languages and is very easy to use. Using `if...else` statements, we will perform condition checking in Rust.

Getting ready

We will require the Rust compiler and any text editor for coding.

How to do it...

Perform the following steps:

1. Create a file named `condition.rs` and enter the following code to the script:

```
use std::i32;
fn main() {
    let age : i32 = 10;
    // If else statements
    if age <= 18{
        println!("Go to School");
    } else if (age > 18) && (age <= 28){
        println!("Go to college");
    } else {
        println!("Do something with your life");
    }

    // if/ else statement in one line
    let can_vote = if (age >= 18) {true} else
    {false};
    println!("Can vote {}", can_vote );
}
```

2. Create a variable named `age` and assign it to an integer with the value `10`.
3. The preceding code has an `if...else` statement to make a decision about the `age` value. It performs print operations based on the conditions.

You should get the following screenshot as output upon running the preceding code:

```
[vigneshwars-MacBook-Air:chapter2 vigneshwerdhinakaran]$ rustc -A warnings condition.rs
[vigneshwars-MacBook-Air:chapter2 vigneshwerdhinakaran]$ ./condition
Go to School
Can vote false
```


How it works...

In this recipe, we implemented an `if...else` statement to perform conditional statements in Rust. The conditions are performed in the `age` variable. In this recipe, we assigned an immutable variable taking the value `10`; after this, we compared it with various rules and performed an action based on the qualifying rule.

These rules are the conditions that the developer generates in the form of a mathematical operation that yields a result of `true` or `false`. Based on the output of the operation, we select a particular set of actions inside the scope of the decision statement.



The `if...else` statements are a great tool for developers to route the program logic. They are ideal for comparing thresholds at the end state of the application for making a logical decision.

In the preceding case, we checked three cases in the following flow:

- The `if` statement checks whether the `age` variable is less than `18`. If the operation returns `true`, then we go ahead and print `Go to School`.
- The next condition is checked in the `else...if` statement when the first condition returns `false`; here we check whether the age is between `18` and `28`, and if this condition returns `true`, we print `Go to college`.
- Lastly, we have the `else` statement, which has no condition and is executed only when all the preceding conditions fail.

It's often a very important skill to write in a very optimized manner. We should learn the ability to develop the skill of writing less and optimized code.

The preceding set of statements contains a lot of lines of code, but we can write it in an optimized way, where we would have the `if...else` statement

along with the condition in a single line. The general syntax for this case is as follows:

```
| let variable = if (condition 1) {true} else {false};
```

We have a variable to which we assign the `if` the `condition 1` operation produces `true`; alternatively, we assign the value from the `else` statement.

Looping operations in Rust

In this recipe, you will learn about looping statements in Rust. Looping statements that we are referring to in Rust provide interactive functionality. Using the `loop`, `while`, and `for` keywords, we can perform iterative operations in Rust.

Getting ready

We will require the Rust compiler and any text editor for coding.

How to do it...

Perform the following steps:

1. Create a file named `looping.rs` and enter the following code in the script.
2. In the `main` function, perform a looping operation on the mutable variable `x`, which is initially assigned to the integer value of `1`.
3. Define a `loop` statement, which is an infinite iterative statement, and check the various conditions inside its scope:

```
fn main() {  
  
    // mutable variable whose value can be changed  
    let mut x = 1;  
    println!(" Loop even numbers ");  
  
    // Continously loops  
    loop {  
        // Check if x is an even number or not  
        if (x % 2 == 0){  
            println!("{}",x);  
            x += 1;  
            // goes to the loop again  
            continue;  
        }  
        // exit if the number is greater than 10  
        if (x > 10) {  
            break;  
        }  
        // increment the number when not even  
        x+=1;  
    }  
}
```

4. Create a mutable variable `y` and assign it to the integer value `1`, and define a `while` loop with the `y < 10` condition:

```
let mut y = 1;  
// while loop  
println!("while 1 to 9 ");  
while y < 10 {  
    println!("{}",y );  
    y +=1;  
}
```

5. Perform a similar operation as for the `while` loop. Here, use the `for` loop to iterate over the 1 to 9 range on the mutable variable `z`, which is initially assigned to the value `1`:

```
let mut z = 1;
//for loop
println!(" For 1 to 9");
for z in 1 .. 10 {
    println!("{}" ,z );
}
```

You should get the following screenshot as output upon running the preceding code:

```
[vigneshwers-MacBook-Air:chapter2 vigneshwerdhinakaran$ rustc -A warnings looping.rs
[vigneshwers-MacBook-Air:chapter2 vigneshwerdhinakaran$ ./looping
Loop even numbers
2
4
6
8
10
while 1 to 9
1
2
3
4
5
6
7
8
9
For 1 to 9
1
2
3
4
5
6
7
8
9
```


How it works...

The `loop` is an iterative keyword in Rust where the statements inside its scope run forever, that is, indefinitely, unless they are explicitly stopped by the `break` statement. This is very useful when we want a process a particular task in an application until it reaches a particular state for further processing. Consider a video storage application where I want to continuously save the camera feeds until the users give a command to stop the application.

In this recipe, we declared a mutable `int` variable `x`, which we initialized with the value `1`. When it enters the `loop` statement, we had two conditions for it. The first condition prints the value of `x`. Only when it is an even number, we use the `%` operator to perform this divisibility operation, followed by an increase in the value.

Then we used the `continue` keyword, which goes back to the `loop`. The preceding statements of the keyword will not be executed. The second condition checks whether the value of `x` is greater than `10`. This condition will only be reached at runtime. When the value of `x` is odd in this case, we break the `loop`, which is the exit point of the infinite loop, which is similar to the case of the stop button in the video application discussed in the preceding example. Next, we increase the value of the next iteration.

While printing `1` to `9` in two different ways, the first method uses `while`, where we have a condition placed which is at first compared to the loop that does not have a condition. All the while, the loop checks the condition at every iteration. Only if it is `true`, it proceeds. In the preceding case, we had an immutable variable `y`, which was initialized with the value `1`. We had a condition which checks whether `y` is less than `10` at every iteration. In each iteration, we print the value of `y` and increase its value by `1`.

The second way to do the preceding activity is by using the `for` looping statement, where we specify a range of values in which we want to operate. We don't have any explicit condition checking, as in the case of other

looping statements. We declared an immutable variable `z`, which was initialized to the value `1` and then iterated from `1` to `10` in the loop where we print the value in every step.

Looping statements are really handy to the developer when there is a requirement to perform a particular task repeatedly in the application.

Defining the enum type

In this recipe, you will learn about using the `enum` type in Rust. In Rust, the `enum` type lets the developer represent data in multiple formats, and each format can optionally have specific data associated with it. Using `enum` keywords, we perform iterative operations in Rust.

Getting ready

We will require the Rust compiler and any text editor for coding.

How to do it...

Perform the following steps:

1. Create a file named `enum.rs` and enter the following code in the script:

```
fn main() {  
  
    let hulk = Hero::Strong(100);  
    let fasty = Hero::Fast;  
    //converting from  
    let spiderman = Hero::Info  
    {name:"spiderman".to_owned(),secret:"peter  
parker".to_owned()};  
    get_info(spiderman);  
    get_info(hulk);  
    get_info(fasty);  
}
```

2. Declare an `enum` date type, namely `Hero`:

```
// declaring the enum  
enum Hero {  
    Fast,  
    Strong(i32),  
    Info {name : String, secret : String}  
}
```

3. Create a function named `get_info` that will take the `enum` data type as an argument:

```
// function to perform for each types  
fn get_info(h:Hero){  
match h {  
    Hero::Fast => println!("Fast"),  
    Hero::Strong(i) => println!("Lifts {} tons",i ),  
    Hero::Info {name,secret} => { println!(" name is: {} secret is  
: {}", name,secret); } ,  
}
```

You should get the following screenshot as output upon running the preceding code:

```
[vigneshwers-MacBook-Air:chapter2 vigneshwerdhinakaran]$ rustc -A warnings enum.rs  
[vigneshwers-MacBook-Air:chapter2 vigneshwerdhinakaran]$ ./enum  
 name is : spiderman secret is : peter parker  
 Lifts 100 tons  
 Fast
```


How it works...

The `enum` is a very important Rust type as it allows a particular data type to be associated with multiple data variants. A value of an `enum` type contains information about which data variant it is associated with.

Another important point to note about `enum` in Rust before moving to the code explanation is you can use the `::` syntax in order to use the name of each data variant and assign values to the variables.

In the recipe, we created an `enum` type `Hero`, which has three types of data variants: `Fast`, which is not specified with explicit data requirements; `strong(i32)`, which requires a 32-bit integer input; and `Info`, which supports two string data variables, `name` and `secret`.

Next up, let's check out the initialization of these data variants in the `main` function. Here, we have created three variables that represent the three data variants and initialized them with the required data requirement. We also called the `get_info()` function three times by passing different `enum` data variants to print the data values.

The `fast` is initialized with the `Hero::Strong(100)` `enum` type, `fasty` with `Hero::Fast`, and `spiderman` with `Hero::Info`, which requires two variables:
`name:"spiderman".to_owned()` and `secret:"peter parker".to_owned()`.



Note that while declaring the values to `Hero` data variant `Info`, we assigned data variables with strings along with `.to_owned()` method, this is done in order to ensure the string is owned when borrowed, as `&str` is an immutable reference to a string and using `to_owned()` turns it into a string that we own.

The `get_info(argument : enum type)` function takes the `enum` as the data type, and when we pass each of the different data variants, the arguments are assigned with those values. Then we used the `match` statement, which is a decision-

making statement, to compare the argument with the different types of data variants mentioned as the different cases in the `match` statement.

We passed the `fast` variable, which is of the type `Fast`--variant of `Hero`--and it will print `Fast`, which is the first case of the `match` statement. Similarly, for the `spiderman` case and `hulk`, which are of the types `Info` and `Strong`, respectively, the corresponding statement in the match of the `get_info` function will be executed.

Defining closures

At an uber level, closures are similar to functions, and calling a closure is exactly like a function. Closures are similar to **lambdas**, which are basically functions that operate over the variables in a closed scope.

Getting ready

We will require the Rust compiler and any text editor for coding.

How to do it...

Perform the following steps:

1. Create a file named `closures.rs` and enter the following code in the script:

```
| use std::i32;
| fn main() {
```

2. Define a closure and name it `sum_num`:

```
| // define a closure
| let sum_num = |x:i32, y:i32| x+y;
| println!("7 + 8 ={}", sum_num(7, 8));
```

3. Create another closure, namely `add_ten`:

```
| // example 2
| let num_ten = 10;
| let add_ten = |x:i32| x+num_ten;
| println!("3 + 10 ={}", add_ten(3));
| }
```

We should get the following screenshot as output upon running the preceding code:

```
[vigneshwers-MacBook-Air:chapter2 vigneshwerdhinakaran$ rustc -A warnings closures.rs
[vigneshwers-MacBook-Air:chapter2 vigneshwerdhinakaran$ ./closures
7 + 8 =15
3 + 10 =13
```


How it works...

An important thing about a closure is that its bound or its operations are within a scope where it is defined. It is similar to a function that uses the free variable in the environment of its operation.



Closures are great ways to write mathematical operations. If the developer is working on Rust to speed up the mathematical computation of the application, then the developer can maintain closures in his or her code for different equations for better optimization, code debugging, and benchmarking.

In this recipe, we created two closures in the `main` function. The basic way to create a simple closure is to have a variable assigned to an operation, before in which we could declare the variable types in the pipe symbol in a `let` statement. The first closure is named `sum_num`, which basically adds two numbers and returns an integer output as the two variables it uses, namely `x` and `y`, which are 32-bit integers. The second closure `add_ten` adds a fixed integer value of `10` to the integer that is passed to the closure. Calling a closure is similar to that of a function. The convention is to call the name of the closure followed by the parameters to be passed to the closure operation. In this recipe, we called `sum_num(7, 8)`, which gave the output of `15` at runtime and `add_ten(3)`, which produced `13`.

Performing pointer operations in Rust

Rust provides different smart pointers. These are different types of pointers used in Rust for different use cases, but `&mut T` is a mutable (exclusive) reference that is one of the operations.

Getting ready

We will require the Rust compiler and any text editor for coding.

How to do it...

Perform the following steps:

1. Create a file named `pointer.rs` and enter the following code in the script:

```
| use std::i32;
| fn main() {
```

2. Create a vector named `vect1` and assign it to `vec![1, 2, 3]`:

```
| let vect1 = vec![1, 2, 3];
|
| // Error in case you are doing this in case of non primitive
| value
| // let vec2 = vect1
| // println!("vect1[0] : {:?}", vect1[0]);
|
| let prim_val = 1;
| let prim_val2 = prim_val;
| println!("primitive value :- {}", prim_val);
```

3. Pass `&vect1` to the `sum_vects()` function:

```
| // passing the ownership to the function
| println!("Sum of vects : {}", sum_vects(&vect1));
| // Able to pass the non primitive data type
| println!("vector 1 {:?}", vect1);
| }
```

4. Perform summation operations for each value of the vector:

```
| // Added a reference in the argument
| fn sum_vects (v1: &Vec<i32>) -> i32 {
|     // apply a closure and iterator
|     let sum = v1.iter().fold(0, |mut sum, &x| {sum += x; sum});
|     return sum;
| }
```

You should get the following screenshot as output upon running the preceding code:

```
[vigneshwers-MacBook-Air:chapter2 vigneshwerdhinakaran$ rustc -A warnings pointer.rs
[vigneshwers-MacBook-Air:chapter2 vigneshwerdhinakaran$ ./pointer
primitive value :- 1
Sum of vects : 6
vector 1 [1, 2, 3]
```


How it works...

Ownership and borrowing are the main concepts on which Rust is built, and the standard APIs given by Rust are based on this concept. In the preceding snippet, we created a vector, namely `vect1`, and assigned it `1, 2, 3` using the `vec!` keyword.

Note that a vector is a non-primitive value. Vectors cannot be reused after, as shown in the commented section of the code. The compiler will throw an error saying that `vect1` is a moved value and cannot be used. This is the case when we assign `vect1` to `vect2` and try to assign `vect1` to the print statement.

In the `sum_vecs(&vect1)` function, we passed the ownership of `vect1` to the `sum_vector` function, which iterates through each of the objects of the vector and produces the sum. Note that we passed `vect1` with a `&` symbol. This way, we shared the vector as a reference or pointer, but if we had passed it as `&mut vect1`, then the function would have had the ability to mutate or make changes in the values of the vector. We verify this by printing `vect1` after processing it from the `sum_vecs` function, which still yields the same result.

In `sum_vecs(&vect1)`, we had `v1`, which is the argument to which `vect1` is moved to. The vector has a method from the standard APIs that allows the `iter` function to read one data object from the zero position.



The `fold()` function takes two arguments: an initial value and a closure. The closure again takes two arguments: an accumulator and an element. The closure returns the value that the accumulator should have for the next iteration.

Here the accumulator is `sum` and the element is `x`, which is added to `sum` in every iteration. Note that `x` being mutable in the definition of the closure can change the values in the scope of its operation. This is stored in the `sum` variable and returned to the `main` function.

Defining your first user-defined data type

In this recipe, you will learn about structs, which is a way in which you can create complex data types in Rust. Using `struct`, we will define user-defined data types in Rust.

Getting ready

We will require the Rust compiler and any text editor for coding.

How to do it...

Perform the following steps:

1. Create a file named `struct.rs` and enter the following code in the script:

```
use std::f64;
fn main() {

    // create a struct variable
    let mut circle1 = Circle {
        x:10.0, radius : 10.0
    };

    // print radius and variable x
    println!("x:{}, radius : {}", circle1.x,
             circle1.radius );
    println!("Radius : {}", get_radius(&circle1) );
}
```

2. Create a `struct` named `Circle` with two parameters, namely `x` and `radius`:

```
// define your custom user data type
struct Circle {
    x : f64,
    radius : f64,
}
```

3. Define a function `get_radius` by accepting `Circle` as a user-defined data type:

```
// get radius function
fn get_radius(c1 : &Circle) -> f64{
    c1.radius
}
```

You should get the following screenshot as output upon running the preceding code:

```
[vigneshwers-MacBook-Air:chapter2 vigneshwerdhinakaran$ rustc -A warnings struct.rs
[vigneshwers-MacBook-Air:chapter2 vigneshwerdhinakaran$ ./struct
x:10, radius : 10
Radius : 10
```


How it works...

At some point of the product development life cycle, developers often have too many variables to handle and the code becomes really complex. This is where structs appear as a big savior. Structs enable developers to create complex data types, where they allow the unification of multiple data types under a single name.

In this recipe, we created a custom data type named `circle`, which has two labels `radius` and `x` of the type `f64`, which is a 64-bit `float` type. Both the parameters here are related to the `circle` data type and uniquely express their features.



Consider use cases such as database management, machine learning models, and so on, where the developer has to handle multiple variables conveying the property of a single task/entity. Structs, in these cases, are great tools to utilize for making the code more optimized and modular. This makes the life of a developer easy; we can debug errors easily and scale up features on requests of the application/product.

We use the `struct` keyword to create a user-defined data type, where the custom name is provided after the keyword but along with the types of the different labels or variables it uses.

In the `main` function, we initialized a mutable variable `circle1` of the user-defined data type `circle` and populated it with its required values, which are `10.0` for `radius` and `10.0` for `x`. We did this to access the variable in the scope of the program. We get the value by calling the variable name label we require, that is, we get the value of the assigned values by calling `circle1.x` and `circle.radius`.

We pass the reference of `circle1` to `get_radius`, where we have an argument `c1` of the data type `circle` from which we get the radius of `c1.radius`. Then, we call the function with `get_radius(&circle1)` to get the value.

Adding functionality to the user-defined data type

You will learn about performing method calls using the `impl` keyword in Rust, which helps in adding functionality to a user-defined data type. In this recipe, the `impl` block helps us create the methods.

Getting ready

We will require the Rust compiler and any text editor for coding.

How to do it...

Perform the following steps:

1. Create a file named `implement.rs` and enter the following code in the script:

```
use std::f64;

fn main() {
    // create a struct variable
    let mut circle1 = Circle {
        x:10.0, radius : 10.0
    };
    println!("x:{}, radius : {}", circle1.x,
            circle1.radius );
    println!("x : {}", circle1.get_x());
}
```

2. Create a `struct` named `Circle` with two parameters, `x` and `radius`:

```
// define your custom user data type
struct Circle {
    x : f64,
    radius : f64,
}
```

3. Create the `get_x` method for the user-defined `Circle` data type:

```
// recommended way of creating structs
impl Circle {
    // pub makes this function public which makes it
    // accessible outside the scope {}
    pub fn get_x(&self) -> f64 {
        self.x
    }
}
```

You should get the following screenshot as output upon running the preceding code:

```
[vigneshwers-MacBook-Air:chapter2 vigneshwerdhinakaran]$ rustc -A warnings implement.rs
[vigneshwers-MacBook-Air:chapter2 vigneshwerdhinakaran]$ ./implement
x:10, radius : 10
x : 10
```


How it works...

In this recipe, we created a custom data type named `circle`, which has two labels: `radius` and `x` of the type `f64`, which is a 64-bit `float` type. Both the parameters here are related to the `circle` data type and uniquely express its features.

In the `main` function, we initialized a mutable variable `circle1` of the user-defined data type `circle` and populated it with its required values, which are `10.0` for `radius` and `10.0` for `x`. To access the variable in the scope of the program, we get the value by calling the variable name label we require, that is, we get the value of the assigned values by calling `circle1.x` and `circle.radius`.

But, we went ahead and created unique functionalities for each data type so that they can perform a unique operation on the labels associated with them; this eliminates the need to pass argument values to externally created functions. We used `impl` to achieve this method call, where we defined functionalities for the data type.

This feature allows the developer to call functions of the data type using `datatype_name.function1().function2()`, which reduces the function call complexity and delivers optimized code.

In the `main` function, we call `circle1.get_x()` to get the value of the `x` value. If you closely observe the `impl` code part of `Circle`, you'll notice we passed `&self` to the `get_x()` method, which is a reference to the `circle` label's data type.

Similar functionality for different data type

You will learn about the `trait` feature of Rust in this recipe is similar to `impl`, which helps the developer make a method call of the user-defined data type. However, `trait` provides many more features, such as inheritance and control, over the functionality that the user-defined data type provides.

Getting ready

We will require the Rust compiler and any text editor for coding.

How to do it...

Perform the following steps:

1. Create a file named `trait.rs` and enter the following code in the script:

```
use std::f64;
fn main() {
    // variable of circle data type
    let mut circle1 = Circle {
        r: 10.0
    };
    println!("Area of circle {}", circle1.area());
}

// variable of rectangle data type
let mut rect = Rectangle {
    h: 10.0, b: 10.0
};
println!("Area of rectangle {}", rect.area());
```

2. Create a `struct` named `Rectangle` with the parameters `h` and `b`, both 64-bit `float` data types:

```
// userdefined data type rectangle
struct Rectangle {
    h: f64,
    b: f64,
}
```

3. Create a `struct` named `Circle` with the parameter `r`, which is a 64-bit `float` data type:

```
// userdefined data type circle
struct Circle {
    r: f64,
}
```

4. Create a `trait` named `HasArea` with the `area` functionality:

```
// create a functionality for the data types
trait HasArea {
    fn area(&self) -> f64;
}
```

5. Define the `area` function for the `Circle` user-defined data type:

```
// implement area for circle
impl HasArea for Circle {
```

```
| fn area(&self) -> f64 {  
|     3.14 * (self.r * self.r)  
| }  
| }
```

6. Define the `area` function for the `Rectangle` user-defined data type:

```
| // implement area for rectangle  
| impl HasArea for Rectangle {  
|     fn area(&self) -> f64 {  
|         self.h * self.b  
|     }  
| }
```

You should get the following output upon running the preceding code:

```
[vigneshwers-MacBook-Air:chapter2 vigneshwerdhinakaran$ rustc -A warnings trait.rs  
[vigneshwers-MacBook-Air:chapter2 vigneshwerdhinakaran$ ./trait  
Area of circle 314  
Area of rectangle 100
```


How it works...

In this recipe, we applied all the concepts that we learned in the previous ones. We created two `struct` types: `Circle` with a `radius` of `f64` and `Rectangle` with the parameters `h` and `b` of `f64`. Then, we created the `area` functionality for each `struct` data type that operates on the data of the labels, as they are referenced by `self`.

The function definition of both user-defined data types is different in terms of the mathematical operation. We defined the data type `Circle` and `Rectangle` in the `main` function. We called the functions in real time by `Circle.area()` and `Rectangle.area()`.

Here, we observe that both data types provide a similar kind of functionality; this is where the `trait` comes into place. It basically tells the compiler the functionality that a particular function would use, so we implement the `trait`. For the data type in this recipe, we have a `trait` named `HasArea`, which contains only the signature of the function that is inside the scope, which contains the output that is returned and the reference which was passed as the argument. In this recipe, we had a signature of `fn area(&self) -> f64;`, which indicated the output of the computation in a 64-bit `float` type. The function operates by taking a reference to the label and values of the data type.

Deep Diving into Cargo

In this chapter, we will cover the following recipes:

- Creating a new project using Cargo
- Downloading an external crate from crates.io
- Working on existing Cargo projects
- Running tests with Cargo
- Configuration management of the project
- Building the project on the Travis CI
- Uploading to crates.io

Introduction

Cargo is one of the unique selling points of Rust, which is the first of its kind in the system programming space. Cargo is Rust's package manager, and it makes a developer's life easy in regard to creating, developing, packaging, maintaining, testing, and deploying application code or tools to production, without great effort. In this chapter, we will cover recipes that will enable a developer to utilize all the features of Cargo and make a production-grade Rust application from day one of development.

Creating a new project using Cargo

Cargo is a unique offering from Rust and is very new in the sphere of system programming. It is also one of the selling points of Rust, as it enables developers to package, ship, and test their Rust application.

We will cover a lot of functionalities of Cargo in this chapter.

Getting ready

We require the Rust compiler, Cargo, and any text editor for coding.

How to do it...

1. Open the Terminal.
2. Go to the directory where you want to create the project:

```
| cd project_location
```

3. Enter the following command to create a new Rust project:

```
| cargo new project_name --bin
```

Create a project named `hello_world`, as shown in the following example:

```
| cargo new hello_world --bin
```

You should get the following output:

```
viki@Vigneshwer:~/rust_cookbook/chapter3$ cargo new hello_world --bin
    Created binary (application) `hello_world` project
```

4. You should have a new folder created with the name of the project.

First, get into the project and examine it:

```
| cd hello_world
| tree .
```

You should get the following output:

```
viki@Vigneshwer:~/rust_cookbook/chapter3$ cd hello_world/
viki@Vigneshwer:~/rust_cookbook/chapter3/hello_world$ tree .
.
|-- Cargo.toml
|-- src
   '-- main.rs

1 directory, 2 files
```

This is the whole structure of the newly created project.

5. Print the content of the `Cargo.toml` file using the `cat` command in Linux:

```
| cat Cargo.toml
```

You should get the following output:

```
Cargo.toml  src
viki@Vigneshwer:~/rust_cookbook/chapter3/hello_world$ cat Cargo.toml
[package]
name = "hello_world"
version = "0.1.0"
authors = ["Vigneshwer.D <dvigneshwer@xyz.com>"]

[dependencies]
```

6. Go to the `src` directory inside the project where you will find the default `main.rs` file and print its content using the `cat` command in Linux:

```
|   cd src
|   cat main.rs
```

You should get the following output:

```
viki@Vigneshwer:~/rust_cookbook/chapter3/hello_world$ cd src/
viki@Vigneshwer:~/rust_cookbook/chapter3/hello_world/src$ ls
main.rs
viki@Vigneshwer:~/rust_cookbook/chapter3/hello_world/src$ cat main.rs
fn main() {
    println!("Hello, world!");
}
```

7. Build the sample project that comes with the `cargo new` command:

```
|   cargo build
```

You should get the following output:

```
viki@Vigneshwer:~/rust_cookbook/chapter3/hello_world/src$ cargo build
  Compiling hello_world v0.1.0 (file:///home/viki/rust_cookbook/chapter3/hello_world)
  Finished debug [unoptimized + debuginfo] target(s) in 1.32 secs
```

8. The Cargo build will create a new directory named `target` and a file named `cargo.lock`.

Run the compiled project, which is a executable file present in the target directory.

```
|   . project_directory/target/debug/hello_world
```

Since this is an executable, you should be able to see the output of the application.

You should get the following output:

```
viki@Vigneshwer:~/rust_cookbook/chapter3/hello_world/src$ cd ..
viki@Vigneshwer:~/rust_cookbook/chapter3/hello_world$ ls
Cargo.lock  Cargo.toml  src  target
viki@Vigneshwer:~/rust_cookbook/chapter3/hello_world$ ./target/debug/hello_world
Hello, world!
```

9. We have to follow the preceding two steps in order to build the executable code and then execute the Rust application, but with the `cargo run` command we can perform both simultaneously:

```
|   cd project_location
|   cargo run
```

You should get the following output:

```
viki@Vigneshwer:~/rust_cookbook/chapter3/hello_world$ cargo run
    Finished debug [unoptimized + debuginfo] target(s) in 0.0 secs
        Running `target/debug/hello_world'
Hello, world!
```


How it works...

Cargo is a command-line tool that comes with the installation of Rust. It's essential for the following reasons:

- It introduces metadata files that convey all the details about the project
- It calls the `rustc` compiler to build the project
- It helps the developer structure and maintain the code better

The `cargo new` commands create the complete folder structure for us. For our project, the `--bin` indicates the binary file. This means we are creating a Rust application that is expected to work out of the box in regard to solving a real-world problem. However, in this case, we created a library that would not use the `--bin` option as a command-line tool. Libraries are known as crates in Rust. We will create a crate in Rust later, in the upcoming chapters.

`Cargo.toml` is a manifest file that contains all the metadata that Cargo needs to compile your project. When you run the `cargo build` command, you will see that the source code is converted into an executable byte code, which would be the end application; this creates the `target` directory and places the executable in the `debug` folder inside it. Inside the `debug` folder, we mainly have `deps` directory which contains the different dependent crates which were downloaded to execute the application.

Your project can optionally contain folders named `example`, `test`, and `bench`, which Cargo will treat as containing examples, integration tests, and benchmarks respectively.

Rust is very smart and only compiles when there are changes in the code.



Compiling `cargo build --release` in debug mode is suitable for development, and its compilation time is shorter since the compiler does not do any optimizations and checks. However, when you run the code in release mode, it will take longer to compile, but the code will run faster in production. The

release mode will prepare the build in the `release` folder inside target instead of `debug` directory.

We see that the build process created a `Cargo.lock` file. This file contains all the information about our dependencies. We will cover this file in detail in an upcoming recipe.

In order to compile multiple binaries in the same Rust project, we have to make certain entries in the `Cargo.toml` file where we explicitly mention the target that we want to build. By default, Cargo compiles the `main.rs` file in the `src` folder with the same name of the project, but for compiling multiple binaries, say `daemon` and `client`, which need to be built, we make the following mention changes in the `Cargo.toml` file:

```
[[bin]]  
name = "daemon"  
path = "src/daemon/bin/main.rs"  
  
[[bin]]  
name = "client"  
path = "src/client/bin/main.rs"
```

This would build an additional two binaries named `daemon` and `client`, along with the project binary.



Similarly we can have sections such as `[lib]`, `[[bench]]`, `[[test]]`, and `[[example]]` in a configuration file to build libraries, benchmarks, tests, and examples.

Downloading an external crate from crates.io

To create complex applications for solving real-world problems, we need to reuse other open source projects and dependencies for faster development.

The <https://crates.io/> is the Rust community's central repository that serves as a location for discovering and downloading packages. The command-line tool `cargo` is configured to find requested packages and download and use them. You will learn how to download and maintain external crates (dependencies) in this recipe.

Getting ready

We require the Rust compiler, Cargo, and any text editor for coding.

How to do it...

1. Open the `cargo.toml` file in your favorite text editor; in this recipe, we will use the `nano` editor:

```
| nano Cargo.toml
```

You should get the following output:

```
viki@Vigneshwer:~/rust_cookbook/chapter3/hello_world$ nano Cargo.toml
```

2. Add a `[dependencies]` section to the `cargo.toml` file and enter `time = "0.1.12"` and `regex = "0.1.41"` below it.

You should get the following output:

```
[package]
name = "hello_world"
version = "0.1.0"
authors = ["Vigneshwer.D <dvigneshwer@gmail.com>"]

[dependencies]
time = "0.1.12"
regex = "0.1.41"
```

3. Use the `cat` command to see the configuration list:

```
| cat Cargo.toml
```

You should get the following output:

```
viki@Vigneshwer:~/rust_cookbook/chapter3/hello_world$ cat Cargo.toml
[package]
name = "hello_world"
version = "0.1.0"
authors = ["Vigneshwer.D <dvigneshwer@gmail.com>"]

[dependencies]
time = "0.1.12"
regex = "0.1.41"
```

4. Build the project to pull the dependencies from <https://crates.io/>:

```
| cargo build
```

You should get the following output:

```
viki@Vigneshwer:~/rust_cookbook/chapter3/hello_world$ cargo build
  Updating registry `https://github.com/rust-lang/crates.io-index'
    Downloading regex v0.1.80
    Downloading time v0.1.36
    Downloading aho-corasick v0.5.3
    Downloading memchr v0.1.11
    Downloading thread_local v0.2.7
    Downloading utf8-ranges v0.1.3
    Downloading regex-syntax v0.3.9
    Downloading thread-id v2.0.0
    Downloading kernel32-sys v0.2.2
    Downloading winapi v0.2.8
    Downloading winapi-build v0.1.1
      Compiling regex-syntax v0.3.9
      Compiling winapi-build v0.1.1
      Compiling winapi v0.2.8
      Compiling utf8-ranges v0.1.3
      Compiling libc v0.2.20
      Compiling kernel32-sys v0.2.2
      Compiling time v0.1.36
      Compiling memchr v0.1.11
      Compiling thread-id v2.0.0
      Compiling thread_local v0.2.7
      Compiling aho-corasick v0.5.3
      Compiling regex v0.1.80
      Compiling hello_world v0.1.0 (file:///home/viki/rust_cookbook/chapter3/hello_world)
    Finished debug [unoptimized + debuginfo] target(s) in 22.25 secs
```

5. Use the existing crates pulled in our project.

Open the `main.rs` file in the `src` directory using `nano` and enter the following code:

```
| nano main.rs
```

You should get the following output:

```
viki@Vigneshwer:~/rust_cookbook/chapter3/hello_world$ cd src/
viki@Vigneshwer:~/rust_cookbook/chapter3/hello_world/src$ ls
main.rs
viki@Vigneshwer:~/rust_cookbook/chapter3/hello_world/src$ nano main.rs
```

```
// Declare the external crate
extern crate regex;
use regex::Regex;
fn main() {
    let check_date = Regex::new(r"\d{4}-\d{2}-\d{2}$").unwrap();
    println!("Did our date match? {}", check_date.is_match("2017-02-01"));
}
```

You should get the following state output:

```
GNU nano 2.2.6                               File: main.rs
extern crate regex;
use regex::Regex;
fn main() {
    let re = Regex::new(r"^\d{4}-\d{2}-\d{2}$").unwrap();
    println!("Did our date match? {}", re.is_match("2014-01-01"));
}
```

6. Compile and run the project:

```
| cargo run
```

You should get the following output:

```
viki@Vigneshwer:~/rust_cookbook/chapter3/hello_world/src$ cargo run
  Compiling hello_world v0.1.0 (file:///home/viki/rust_cookbook/chapter3/hello_world)
    Finished debug [unoptimized + debuginfo] target(s) in 0.75 secs
      Running `'/home/viki/rust_cookbook/chapter3/hello_world/target/debug/hello_world'
Did our date match? true
```


How it works...

We enter the dependencies we require for the project in the `Cargo.toml` file. This file fetches us the package and version mentioned in the <https://crates.io/> central repository. In the preceding recipe, we downloaded the time and regex crates and also mentioned the desired version that we would like to work on.

When we build a project after modifying the `Cargo.toml` file, it downloads all the modules inside the crate of our local development system and makes an entry in the `Cargo.lock` file, which would contain all the details about the downloaded dependencies.



If you are planning to create a library or improve upon an existing library implementation, it is advisable that you check out whether there are any similar ideas or projects implemented in https://crates.io/ to evaluate the value of your open project. All the projects put up on the https://crates.io/ repository are open source projects available on GitHub.

The fun part about using other dependencies is that you get to reuse the available working version of an application or function you would like to use in the project and reduce your project development time.

We used the `extern crate` command in our Rust script to call the downloaded crates.

`extern crate regex` or `crate` and import all its functions inside the modules it has. We then call them in the code by passing our data.

In the preceding code snippet, we explicitly mentioned the need for using the `use` command to call the `Regex` module in the `regex` crate using `regex::Regex` and checking whether the dates match and print the Boolean value in the terminal.

We call the `unwrap` and `is_match` functions to check whether both the strings are the same or not. We return `true` if they are similar and `false` if they are not.

Working on existing Cargo projects

Cargo is a tool that allows the Rust application to declare their various dependencies to ensure that you will be able to recreate the build by following the same dependencies and version.

At a higher level, it offers configuration management to the Rust project and helps in reproducing the development environment. Usually, in other languages, it's a very tedious and time-consuming process to maintain the different dependencies and configure them each and every time we deploy the application in a different environment.

Cargo provides features out of the box which enables developers and project managers to ship/deploy Rust projects/applications very quickly and without much hassle; this is a very big advantage over other languages.

Getting ready

We require the Rust compiler, Cargo, and any text editor for coding.

How to do it...

1. Clone the project from the project repo in this recipe. Close the `rand` crate from GitHub where it is hosted. Do this by typing the following command in the terminal:

```
| git clone https://github.com/rust-lang-nursery/rand.git
```

You should get the following output:

```
viki@Vigneshwer:~/rust_cookbook/chapter3$ git clone https://github.com/rust-lang-nursery/rand.git
Cloning into 'rand'...
remote: Counting objects: 312195, done.
remote: Total 312195 (delta 0), reused 0 (delta 0), pack-reused 312195
Receiving objects: 100% (312195/312195), 81.94 MiB | 901.00 KiB/s, done.
Resolving deltas: 100% (260384/260384), done.
Checking connectivity... done.
```

This would clone the repo from the GitHub to the local system.

2. Enter the newly created project `rand` and check out the complex production-level package details:

```
| cd rand/
| tree .
```

You should get the following output:

```
viki@Vigneshwer:~/rust_cookbook/chapter3$ ls
hello_world  rand
viki@Vigneshwer:~/rust_cookbook/chapter3$ cd rand/
viki@Vigneshwer:~/rust_cookbook/chapter3/rand$ tree .
.
|-- appveyor.yml
|-- benches
|   |-- bench.rs
|   |-- distributions
|       |-- exponential.rs
|       |-- gamma.rs
|       |-- mod.rs
|       '-- normal.rs
|-- Cargo.toml
|-- LICENSE-APACHE
|-- LICENSE-MIT
|-- rand_macros
|   |-- Cargo.toml
|   |-- README.md
|   |-- src
|       '-- lib.rs
|   '-- tests
|       '-- rand_macros.rs
|-- README.md
`-- src
    |-- chacha.rs
    |-- distributions
    |   |-- exponential.rs
    |   |-- gamma.rs
    |   |-- mod.rs
    |   |-- normal.rs
    |   |-- range.rs
    |   '-- ziggurat_tables.rs
    |-- isaac.rs
    |-- lib.rs
    |-- os.rs
    |-- rand_impls.rs
    |-- read.rs
    '-- reseeding.rs

7 directories, 27 files
```

3. Build the project using the Cargo build command:

```
| cargo build
```

You should get the following output:

```
viki@Vigneshwer:~/rust_cookbook/chapter3/rand$ cargo build
warning: unused manifest key: package.categories
          Updating registry `https://github.com/rust-lang/crates.io-index`
          Downloading libc v0.2.20
          Compiling libc v0.2.20
          Compiling rand v0.3.15 (file:///home/viki/rust_cookbook/chapter3/rand)
          Finished debug [unoptimized + debuginfo] target(s) in 4.0 secs
```


How it works...

The preceding three steps would set up a complete Rust project in your local development environment. This is the power of the Cargo command-line tool, which makes the post- development application life cycle so easy for a developer.

Here, we have basically cloned an existing project and built it. Cargo looked into the manifest file to fetch all the metadata that was required to understand the various dependencies of the project and build them.

Running tests with Cargo

Tests are important for complex production-grade applications as they validate the working of a functional unit of the project. Cargo provides Rust with all the testing functionalities that help in unit and integration tests out of the box.

Getting ready

We require the Rust compiler, Cargo, and any text editor for coding.

How to do it...

1. Run the `cargo test` command inside the project:

```
| cargo test
```

You should get the following output:

```
viki@Vigneshwer:~/rust_cookbook/chapter3/hello_world$ cargo test
  Compiling hello_world v0.1.0 (file:///home/viki/rust_cookbook/chapter3/hello_world)
    Finished debug [unoptimized + debuginfo] target(s) in 0.87 secs
      Running target/debug/hello_world-05b232e158745d91

running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured
```

2. Mention the name to run a particular test-- `cargo test test_name`:

```
| cargo test foo
```

You should get the following output:

```
viki@Vigneshwer:~/rust_cookbook/chapter3/hello_world$ cargo test foo
  Finished debug [unoptimized + debuginfo] target(s) in 0.0 secs
      Running target/debug/hello_world-05b232e158745d91

running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured
```


How it works...

Cargo looks for tests to run in two places: they are in the `src` directory, where we can have our unit tests' code placed, and the `tests` directory, where we place the integration tests.

In this recipe, we did not have any tests for the project, but we have a dedicated chapter for tests later, where we will deep dive into the various aspects of testing.

Configuration management of the project

We will thoroughly explore the use of `Cargo.lock` and `Cargo.toml` in this recipe and we'll see how they help in configuration management.

Configuration management here refers to the ability to have version control, which includes uploading and downloading the desired version of your project dependencies.

Getting ready

We require the Rust compiler, Cargo, and any text editor for coding.

How to do it...

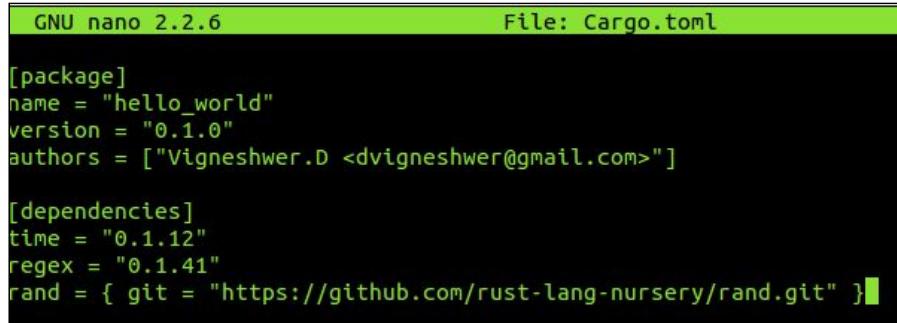
1. Go to the `hello_world` project and enter the GitHub repo link of the `rand` library in the `Cargo.toml` file:

```
| cd hello_world  
| nano Cargo.toml
```

In the following `Cargo.toml`, enter the mentioned dependencies:

```
| rand = { git = "https://github.com/rust-lang-nursery/rand.git"  
| }
```

You should get the following output:



```
GNU nano 2.2.6 File: Cargo.toml

[package]
name = "hello_world"
version = "0.1.0"
authors = ["Vigneshwer.D <dvigneshwer@gmail.com>"]

[dependencies]
time = "0.1.12"
regex = "0.1.41"
rand = { git = "https://github.com/rust-lang-nursery/rand.git" }
```

2. In the same `Cargo.toml` file, enter `rev` and the `SHA` value for the `rand` crate.

You should get the following output:



```
[package]
name = "hello_world"
version = "0.1.0"
authors = ["Vigneshwer.D <dvigneshwer@gmail.com>"]

[dependencies]
time = "0.1.12"
regex = "0.1.41"
rand = { git = "https://github.com/rust-lang-nursery/rand.git", rev = "9f35b8e" }
```

3. Type the `cargo update` command to implement the new changes:

```
| cargo update
```

You should get the following output:

```
vikki@Vigneshwer:~/rust_cookbook/chapter3/hello_world$ cargo update
  Updating registry `https://github.com/rust-lang/crates.io-index`
  Updating git repository `https://github.com/rust-lang-nursery/rand.git`
```

4. Enter `Cargo.lock` and look through the `rand` package details.

You should get the following output:

```
[[package]]
name = "rand"
version = "0.3.14"
source = "git+https://github.com/rust-lang-nursery/rand.git?rev=9f35b8e#9f35b8e439eeedd60b9414c58f389bdc6a3284f9"
dependencies = [
    "libc 0.2.20 (registry+https://github.com/rust-lang/crates.io-index)",
]
```



Never make any manual changes in the `Cargo.lock` file. It is not intended to be edited manually.

How it works...

Cargo parses through the manifest file data, and, based on the data entered, it performs certain tasks.

First, we entered the GitHub repo link of the project in the `Cargo.toml` file to install the particular crate. Since we did not mention any other detail--for example, which version--Cargo will pull all the modules and resources of the latest commit to the master branch of the project.

However, the problem with this approach is that, if the project changes its layout or code tomorrow, our current implementation may be affected when we update the package.

To avoid this, we assigned the `rev` value to the commit ID of the project whose resource we have used for our current build. This seems to solve our problem, but we still have to remember the commit ID, which is error-prone when we enter a wrong ID.

Enter `Cargo.lock` using the `nano` command. Since the `lock` file exists, we don't need to manually keep track of the revision of the dependencies. Cargo will create entries in the `Cargo.lock` file with the details of the revision for each package used in the application.

When you build for the first time, Cargo will take the latest commit and write that information to `Cargo.lock`. Imagine that we are shipping the project to a new host where we want to configure the Rust application. Cargo will use the exact `SHA` from the `Cargo.lock` file even though you should not have mentioned it in the `Cargo.toml` file. We can see there is a lot of information about the package in the `lock` file, which will help us reproduce the same dependencies later.

Building the project on the Travis CI

Travis CI is a continuous integrated software that reads the `yml` file in your project repository. It provides instructions or commands to the tools on how you should build, test, and ship.

In this recipe, you will learn how to build a Rust application using the `travisci` tool whose code is hosted on GitHub.

Getting ready

We require the Rust compiler, Cargo, and any text editor for coding. Host the Rust application in a GitHub repo and integrate the `TravisCI` tool to the GitHub repo.

How to do it...

1. Create a `.travis.yml` file in the project using the `touch` command:

```
| cd project_location  
| touch .travis.yml
```

2. Enter the following configuration for testing the application in different Rust builds by opening the file using any text editor; we are using `nano` in this recipe:

```
| nano .travis.yml  
  
language: rust  
rust:  
  - stable  
  - beta  
  - nightly  
matrix:  
  allow_failures:  
    - rust: nightly
```

You should get the following output:

```
viki@Vigneshwer:~/Documents/events/deeprust$ cat .travis.yml  
language: rust  
rust:  
  - stable  
  - beta  
  - nightly  
matrix:  
  allow_failures:  
    - rust: nightly
```

3. Make a change in the project and push the project to the GitHub repo.



The build process is triggered on TravisCI platform each and every time we push a commit in the project repository. This process is configurable by the advance options available in the TravisCI platform, where you can schedule the build or select repo branches to build from.

You should get the following screenshot as output in the Travis CI platform:

Current Branches Build History Pull Requests

✓ master Update README.md -o #21 passed

Commit a3257d5 Ran for 28 sec
Compare 9ecd199..a3257d5 7 days ago
Branch master

vigneshwer dhinakaran authored GitHub committed

Job log View config

```
> 1 Worker information
> 6 Build system information
73
> 74 $ export DEBIAN_FRONTEND=noninteractive
> 123 $ git clone --depth=50 --branch=master https://github.com/dvigneshwer/deeprust.git dvigneshwer/deeprust
134
135 This job is running on container-based infrastructure, which does not allow use of 'sudo', setuid and setgid executables.
136 If you require sudo, add 'sudo: required' to your .travis.yml
137 See https://docs.travis-ci.com/user/workers/container-based-infrastructure/ for details.
138
139
> 140 Installing Rust
164
165
166
167 $ rustc --version
168 rustc 1.15.1 (021bd294c 2017-02-08)
169 $ cargo --version
170 cargo 0.16.0-nightly (6e0c18c 2017-01-27)
171
> 172 $ cd $TRAVIS_BUILD_DIR/code/deeprust
174 $ cargo build --verbose
175   Compiling deeprust_units v0.1.0 (file:///home/travis/build/dvigneshwer/deeprust/code/deeprust)
176     Running 'rustc --crate-name deeprust_units src/lib.rs --crate-type lib -g -C metadata=f3ab0872f36d51d9 -C extra-filename=-f3ab0872f36d51d9 --out-dir
```


How it works...

Travis CI tools read the instruction specified in the `.travis.yml` file of the project and prepare the environment in which you want to build, test, and release the project.

In the next section, we will break down and understand what each of the commands does in the `build` pipeline:

```
|   language: rust
```

This downloads and installs the latest stable version of the Rust release when setting up the machine. If you only want to test the application in a stable version, you just need the preceding command and there is no need to specify the versions:

```
rust:  
  - stable  
  - beta  
  - nightly
```

For specifying the versions in which you want to test the application, you can use the preceding command where you built the project in the `beta` and `nightly` channels. Even if you are only targeting stable, the Rust team encourages you to test it on other channels as well:

```
matrix:  
  allow_failures:  
    - rust: nightly
```

This will test all three channels, and any breakage in `nightly` will not fail your overall build.

In the preceding `build` logs, if you look closely, you will see the default test script is run by the Travis CI, and it uses the Cargo tool to run your build and test the application using `cargo build` and `cargo test` commands.

Uploading to crates.io

The <https://crates.io/> is a website that acts as a central repository for hosting all the crates made by the amazing Rust community. It contains various projects of various domains.

This helps in instantly publishing the crate and installing them.

Getting ready

We require the Rust compiler, Cargo, and any text editor to code and create an account on the <https://crates.io/> website using the GitHub ID.

How to do it...

1. Go to the project location from where you want to upload your project to <https://crates.io/>:

```
|   cd project_location
```

2. The <https://crates.io/> provides an API token on the accounts setting page when you create a new account or log in with your existing GitHub ID:

```
|   cargo login API_TOKEN
```

3. Package the Rust application:

```
|   cargo package
```

4. Publish the Rust application to <https://crates.io/>:

```
|   cargo publish
```


How it works...

The Cargo login command takes the API token provided and stores it in the `~/.cargo/config` location.



Note that the API token should be kept secret and should not be shared, as it is the way by which we can manage the crates uploaded.

Keep in mind that the name that you used to create the project is going to be the name of the crate, and names of crates are based on a first come, first served basis.

When our project is ready to be packaged, we use the preceding step to create a new folder, named `/target/package`.

The content inside the package is what is going to be uploaded to the <https://crates.io/> website. There will be a file with the naming convention `project_name-version.crate` and a folder `project_name-version`. These names are based on the information given by the developer to `cargo.toml`. We can tweak the value in the configuration file and use the package command until we get the correct name and version number.

In packaging, we have to make sure that we do not upload unnecessary files, such as test script or text files and so on, which were used for testing the library. For this purpose, the `*.crate` file provides a lot of tags or keys that tell about the various features, examples, and resources of the file:

```
[package]
# ...
exclude = [
    "public/assets/*",
    "videos/*",
]
# ...
include = [
    "**/*.rs",
    "Cargo.toml",
]
```

In the preceding section, we explicitly mentioned that all the resources in the `assets` and `videos` folders be excluded; on the other hand, we had a tag named `include` that will help to make sure all the important files are packaged.

The final command, `cargo publish`, looks into the `*.crate` file and publishes the crate in the account created.

Creating Crates and Modules

In this chapter, we will cover the following recipes:

- Defining a module in Rust
- Building a nested module
- Creating a module with struct
- Controlling modules
- Accessing modules
- Creating a file hierarchy
- Building libraries in Rust
- Calling external crates

Introduction

This chapter focuses on introducing modules and crates in Rust. It will help you develop a highly modular and production-grade Rust application. With this, you will have a great file hierarchy, which will compliment the development of features in a modular fashion. The recipes in the chapter will also help you build libraries in Rust and define, control, and access features through external programs.

Defining a module in Rust

All the applications must be modularized so that they become easy to maintain and develop. In Rust, we can have a powerful module system for our application that could hierarchically split the application source code into logical units, which we call modules, and manage their visibility (public/private) across the application.

The dictionary description of a module is that it's a collection of items, such as functions, structs, traits, impl blocks, and even other modules.

You will learn how to create a sample module and understand the concept of visibility in this recipe.

Getting ready

We will require the Rust compiler and any text editor for coding.

How to do it...

1. Create a file named `sample_mod.rs` and open it in your text editor.
2. Write the code header with the relevant information:

```
///- #####  
///- Task: To create a sample module to illustrate  
how to use a module in rust  
///- Author: Vigneshwer.D  
///- Version: 1.0.0  
///- Date: 4 March 17  
///- #####
```

3. Create a module named `sample_mod` using the `mod` keyword and define a function named `private_function` in it:

```
// Defined module named `sample_mod`  
mod sample_mod {  
    // By default all the items in module have private  
    visibility  
  
    fn private_function() {  
        println!("called `sample_mod::private_function()` `  
        \n");  
    }  
}
```

4. Define a function named `sample_function` by marking its visibility as `public`, using the `pub` keyword in the module:

```
// Using the `pub` keyword changes it visibility to public  
pub fn sample_function() {  
    println!("called `sample_mod::sample_function()` ` \n");  
}
```

5. Declare a public function `indirect_private_fn`, which would call `private_function`:

```
// Public items of the module can access the private visible  
items  
pub fn indirect_private_fn() {  
    print!("called `sample_mod::indirect_access()`, that \n ");  
    private_function();  
}
```

6. Define `sample_function` outside the scope of the `sample_mod` module:

```
// Created a sample function to illustrate calling of
fn sample_function() {
    println!("Called the `sample_function()` which is not a part
    of
    mod `sample_mod` \n");
}
```

7. Declare the `main` function, in which we will call each item of the `sample_mod` module to understand how they work and print the output:

```
// Execution of the program starts from here
fn main() {
    // Calling the sample_function which is outside module
    sample_function();

    // Calling the public visible sample_mod's sample_function
    sample_mod::sample_function();

    // Accessing the private function indirectly
    sample_mod::indirect_private_fn();

    // Error! `private_function` is private
    //sample_mod::private_function(); // TODO ^ Try uncommenting
    // this line
}
```

Upon the correct setup of the preceding code, you should get the following screenshot output when you compile and run the program:

```
[vigneshwers-MacBook-Air:chapter4 vigneshwerdhinakaran$ rustc sample_mod.rs
[vigneshwers-MacBook-Air:chapter4 vigneshwerdhinakaran$ ./sample_mod
Called the `sample_function()` which is not a part of mod `sample_mod`
called `sample_mod::sample_function()`

called `sample_mod::indirect_access()`, that
> called `sample_mod::private_function()`
```


How it works...

In this recipe, you learned how to create a sample module in Rust and how you are allowed to call the items of the module.

From this chapter onward we will follow the header style, which is our first step. It basically describes what the code or part of the application unit does. This is a very good code practice to follow, as it helps when another person starts off from where you develop.

We created a module named `sample_mod` using the `mod` keyword, followed by the braces `{}`. The content of the module is its items. Each item is designed to perform a specific task. By default, all the items in the module have private visibility, which means that they cannot be accessed directly outside the scope. In the `sample_mod` module, we explicitly created two functions with public visibility using the `pub` keyword. We added the keyword before creating or declaring the function using the `fn` keyword. This makes the item publicly visible outside the scope of the module. The private function or items can be accessed inside the scope of the module, where all the items can call each other, so we can indirectly call a public item to access a private item from it.

We create four functions in this code, where three are inside the module and one is accessible globally. The first function we created inside `sample_mod` was `private_function`, which, by default, has private visibility. Then we created two public functions, namely `sample_function` and `indirect_private_fn`, where `indirect_private_fn` calls `private_function` in its body.

To call an item of the module outside its scope, we have to follow a particular syntax `--module_name::publically_visible_function name`. In the `main` function, we call `sample_fucntion`, which is a regular function, and the two publicly visible items of the `sample_mod` module: `function sample_mod::sample_function()` and `sample_mod::indirect_private_fn()`. These items will execute the content inside their respective scope.



On calling the private item of the module, it will throw an error saying that the particular item is private. For example, in the preceding recipe, we got an error when we directly called `sample_mod::private_function()`; from the `main` function.

Building a nested module

A nested module is where we want to have modules inside modules, performing different tasks. You will learn how to declare and access the items of a nested module.

Nested modules are a great way of having similar items or functional units in an application together, which helps in maintaining features and debugging crashes.

Getting ready

We will require the Rust compiler and any text editor for coding.

How to do it...

1. Create a file named `sample_nested.rs` in the project workspace
2. Write the code header information, which will provide an overview of the code:

```
//-- #####  
//-- Task: To create a sample nested_mod module  
//-- Author: Vigneshwer.D  
//-- Version: 1.0.0  
//-- Date: 4 March 17  
//-- #####
```

3. Create a module named `sample_mod` using the `mod` keyword:

```
// Defined module named `sample_mod`  
mod sample_mod {
```

4. Create another module named `nested_mod` with `function` and `public` visibility under the `sample_mod` module, which makes `sample_mod` a nested module:

```
// Defined public Nested module named `nested_mod`  
pub mod nested_mod {  
    pub fn function() {  
        println!("called `sample_mod::nested_mod::function()`");  
    }  
}
```

5. Create a function named `private_function` under the `nested_mod` module:

```
#[allow(dead_code)]  
fn private_function() {  
    println!("called  
        `sample_mod::nested_mod::private_function()`");  
}
```

6. Define another module named `private_nested_mod` with a public function named `function` inside `sample_mod`:

```
// Nested modules follow the same rules for visibility  
mod private_nested_mod {  
    #[allow(dead_code)]  
    pub fn function() {  
        println!("called  
            `sample_mod::private_nested_mod::function()`");  
    }  
}
```

```
|    }
```

7. Define the `main` function and call the nested modules with different items declared in them:

```
// Execution starts from main function
fn main() {
    sample_mod::nested_mod::function();
    // Private items of a module cannot be directly accessed, even
    if nested_mod in a public module

    // Error! `private_function` is private
    //sample_mod::nested_mod::private_function(); // TODO ^ Try
    uncommenting this line

    // Error! `private_nested_mod` is a private module
    //sample_mod::private_nested_mod::function(); // TODO ^ Try
    uncommenting this line
}
```

Upon the correct setup of the preceding code, you should get the following output when you compile and run the program:

```
[vigneshwers-MacBook-Air:chapter4 vigneshwerdhinakaran]$ rustc sample_nested.rs
[vigneshwers-MacBook-Air:chapter4 vigneshwerdhinakaran]$ ./sample_nested
called `sample_mod::nested_mod::function()`
```


How it works...

The nested module is a concept where you have a module inside another module. This feature is helpful in having a collection of units of application placed inside a common header.

In this recipe, we created a module named `sample_mod` using the `mod` keyword; in this module, we created two more modules, namely `nested_mod` and `private_nested_mod`, with different visibility. The rules of modules' visibility follow the same rules as those of the items of the modules: we have to explicitly mention the `pub` keyword to mention the visibility of the module. If we do not mention anything, it will be considered private by the Rust compiler.

We then create the items inside the nested modules, which are under the `sample_mod` module. In `nested_mod`, which is a public nested module, we created two items: a public method named `function` and a private method named `private_function`. In the other private nested module `private_nested_mod`, we created a public method named `function`.



We can have the same name for items/units residing inside different modules. In the preceding recipe, we had an item named `function`, which was present in both the nested modules.

In the `main` function, we call the respective items that follow the standard syntax for accessing items. The only difference here is that the items reside inside different nested modules. In this case, we follow the `module_name::nested_module_name:item_name` syntax. Here, we first call the module name, followed by the nested module name and its items.

We call a public nested module a public item, which is `sample_mod::nested_mod::function()`. It will run fine and execute the contents of the item. On calling a private nested module, which in our recipe is `sample_mod::nested_mod::private_function()`, and similar private items of the public nested module, which in our recipe is

`sample_mod::private_nested_mod::function()`, we will get an error mentioning that these items are private, as privately visible units cannot be directly accessed outside the scope.

We have the `#[allow(dead_code)]` attribute for the item `function` in the `private_nested_mod` module. The idea is to disable the `dead_code` lint of the compiler, which will warn about the unused function. In simple terms, lint is software which flags bugs in the code.

Creating a module with struct

This recipe covers the structs that have an extra level of visibility with their fields. The visibility defaults to private and can be overridden with the pub modifier. This visibility only matters when a `struct` is accessed from outside the module, where it is defined and has the goal of hiding information (encapsulation).

Getting ready

We will require the Rust compiler and any text editor for coding.

How to do it...

1. Create a file named `sample_struct.rs` in the project workspace
2. Write the code header with the details of the code:

```
///- #####  
///- Task: To create a sample nested_mod module  
///- Author: Vigneshwer.D  
///- Version: 1.0.0  
///- Date: 4 March 17  
///- #####
```

3. Create a sample module named `sample_struct`, in which you can declare a `public struct` named `WhiteBox`:

```
// Sample module which has struct item  
mod sample_struct {  
    // A public struct with a public field of generic type `T`  
    pub struct WhiteBox<T> {  
        pub information: T,  
    }  
}
```

4. Declare a `public struct` named `BlackBox` with a `private generic type T`:

```
// A public struct with a private field of generic type `T`  
#[allow(dead_code)]  
pub struct BlackBox<T> {  
    information: T,  
}
```

5. Create a `public constructor` named `const_new` using the `impl` keyword, which takes the generic `T` type as input:

```
impl<T> BlackBox<T> {  
    // A public constructor method  
    pub fn const_new(information: T) -> BlackBox<T> {  
        BlackBox {  
            information: information,  
        }  
    }  
}
```

6. Declare the `main` function by calling the `struct` items of the `sample_struct` module, which is the `whitebox struct` item:

```
// Execution starts here  
fn main() {
```

```

// Public structs with public fields can be constructed as
// usual
let white_box = sample_struct::WhiteBox { information:
    "public
     information n" };

// and their fields can be normally accessed.
println!("The white box contains: {} \n",
white_box.information);
// Public structs with private fields cannot be constructed
using field names.
// Error! `BlackBox` has private fields
//let black_box = sample_struct::BlackBox { information:
"classified information" };
// TODO ^ Try uncommenting this line
// However, structs with private fields can be created using

// public constructors
let _black_box = sample_struct::BlackBox::const_new("classified
information \n");

// and the private fields of a public struct cannot be
accessed.
// Error! The `information` field is private
//println!("The black box contains: {}",

black_box.information);
// TODO ^ Try uncommenting this line
}

```

Upon the correct setup of the preceding code, you should get the following output when you compile and run the program:

```

[vigneshwers-MacBook-Air:chapter4 vigneshwerdhinakaran$ rustc sample_struct.rs
[vigneshwers-MacBook-Air:chapter4 vigneshwerdhinakaran$ ./sample_struct
The white box contains: public information

```


How it works...

Until now, in the preceding recipes, we were only looking into modules that had functions that acted as their items. In this recipe, we will create `struct` items that have an extra level of visibility with their fields.

The visibility, by default, is private and can be changed using the `pub` keyword. The visibility allows us to hide information when we try to access a module's, items out of the scope of the module.

We created a module named `sample_struct` using the `mod` keyword. We created two structs with public visibility, named `WhiteBox` and `BlackBox`, using the `pub` and `struct` keywords. In both the `struct` items, we had a generic type `T`.



In Rust, generic means that the particular unit can accept one or more generic type parameters, `<T>`. For example, consider `fn foo<T>(T) { ... }`. Here, `T` is the argument that is specified as a generic type parameter using `<T>`, and it allows it to take any argument of any type.

In both structs, we had a field named `information`, which was tied up with `T`, which is the argument we received. The only difference is that we mentioned `information` in `WhiteBox` as public inside the `struct` and `information` inside `BlackBox` as private by default.

Next up, we created an implementation block for `BlackBox` where we explicitly specified the generic type `T` in the `impl` block. Inside it, we created a method named `const_new`, which we made publically visible, that accepted the generic type `T` as an argument and returned a `BlackBox` struct. The `const_new` acts as a public constructor for `BlackBox`, where we wanted to create the data type.

In the `main` block, we created the `WhiteBox` structure first and assigned it to a variable named `white_box` by `sample_struct::WhiteBox { information: "public information \n" }`. Here, we were calling the module, creating a complex data

structure and printing the `white_box`, information field, which was delivered in the preceding step. Next, we tried to create a variable with the data structure of `BlackBox` in a similar manner. This led to an error saying the field name is private. This is the reason why we created a public method `const_new`, which is a constructor for the `BlackBox` data type. We performed this step by `sample_struct::BlackBox::const_new("classified information \n")` and assigned it to `_black_box`.

This passed the argument from `main` to the `impl` block and created the structure. In this way, we were able to define a public `struct` with private fields, but we were still not able to publically access the information field by `_black_box.information`, as it was a private field originally.

The private members can be accessed by indirect methods in the module. Consider the following code snippet:

```
pub mod root {
    use self::foo::create_foo;
    mod foo {
        pub struct Foo {
            i: i32,
        }
        impl Foo{
            pub fn hello_foo(&self){
                println!("Hello foo");
            }
        }
        pub fn create_foo(i: i32) -> Foo{
            Foo { i: i }
        }
    }
    pub mod bar {
        pub struct Bar {
            pub f: ::root::foo::Foo,
        }
        impl Bar {
            pub fn new(i: i32) -> Self {
                Bar { f: ::root::foo::create_foo(i) }
            }
        }
    }
}
fn main() {
    //still private
    //let f = root::foo::create_foo(42);
    let b = root::bar::Bar::new(42);
    b.f.hello_foo();
}
```

We expose a public constructor `create_foo` in the `foo` module, but the module `foo` still remains private and we only expose `create_foo` in `root` by the `use` keyword, which means that `bar` can now create a `Foo` struct but `create_foo` is still private outside of `root`.

Controlling modules

This recipe focuses on the usage of the `use` keyword in the Rust module, which will help in binding long and tiresome module call paths to a simple single entity. This will improve code readability and provide more control to the developer to call module units. We will also go through the scope of `use` and the concept of shadowing.

Getting ready

We will require the Rust compiler and any text editor for coding.

How to do it...

1. Create a file named `sample_control.rs` in the project workspace.
2. Write the code header with the details of the code:

```
///- #####  
///- Task: To create a sample module to illustrating `use`  
///- Author: Vigneshwer.D  
///- Version: 1.0.0  
///- Date: 4 March 17  
///- #####
```

3. Create `other_function` using the `use` keyword, in order to create the binding for the `deeply` module's items:

```
// Bind the `deeply::nested::function` path to  
// `other_function`.  
use deeply::nested::sample_function as other_function;
```

4. Declare the nested module `deeply` with the nested module named `nested` containing the public function `sample_function`:

```
// Defined a nested  
mod deeply {  
    pub mod nested {  
        pub fn sample_function() {  
            println!("called `deeply::nested::function()` `\\n")  
        }  
    }  
}
```

5. Create a function named `sample_function`:

```
fn sample_function() {  
    println!("called `function()` `\\n")  
}
```

6. Declare the `main` function by calling `other_function`:

```
fn main() {  
    // Easier access to `deeply::nested::function`  
    other_function();
```

7. Create a block. In this block, use the `use` keyword and declare `deeply :: nested :: sample_function`, which is equivalent to binding it to `sample_function`:

```
    println!("Entering a block n");
{
    // This is equivalent to `use deeply::nested::sample_function
    // as sample_function`.
    // This `sample_function()` will shadow the outer one.
    use deeply::nested::sample_function;
    sample_function();

    // `use` bindings have a local scope. In this case, the
    // shadowing of `function()` is only in this block.
    println!("Leaving the block \n");
}
```

8. Call `sample_function` outside the block that is created:

```
sample_function();
}
```

Upon the correct setup of the preceding code, you should get the following output when you compile and run the program:

```
[vigneshwers-MacBook-Air:chapter4 vigneshwerdhinakaran$ rustc sample_control.rs
[vigneshwers-MacBook-Air:chapter4 vigneshwerdhinakaran$ ./sample_control
called `deeply::nested::function()'

Entering a block
called `deeply::nested::function()'

Leaving the block
called `function()'
```


How it works...

We focused on the `use` keyword in this recipe, which is really handy when your modules get deeper (which means there are a lot of nested modules and units). In short, `use` helps bind a long module call to a single name. In the preceding recipe, we had a nested module named `deeply`, where `nested` is the module inside `deeply` that has a public function named `sample_function`. Conventionally, we can call this function by `deeply::nested::sample_function`; however, using the `use` keyword, we can bind this to a single entity and call it in the `main` function, which provides much easier access. Here, we bound this path to `other_function()` and we also created a normal function named `sample_function` to understand the concept of shadowing.

We created a block inside the `main` function and explicitly mentioned `use deeply::nested::sample_function`. We also called `sample_function` after that. This calls the `deeply` module's `sample_function` item rather than the globally available function and the `use` mentioned in the block ends once it goes outside the scope. Calling `sample_function` outside the block will lead to calling the global function, as `use` is not active anymore.

Accessing modules

In this recipe, we will use the `self` and `super` keywords in Rust to provide better access to the module's units and learn about the scope of Rust units. In this recipe, we will create various module units across the code with similar names, to create ambiguity in the unit name. We will check out how the `self` and `super` keywords help the developer overcome these problems.

Getting ready

We will require the Rust compiler and any text editor for coding. Also, create a file named `sample_module.rs` in the project workspace.

How to do it...

1. Create a file named `sample_access.rs` in the project workspace.
2. Write the code header with the details of the code:

```
///- #####  
///- Task: To create a sample module to illustrating `self` and  
`super`  
///- Author: Vigneshwer.D  
///- Version: 1.0.0  
///- Date: 4 March 17  
///- #####
```

3. Create a function named `sample_function` and print "called `sample_function()`" in its scope:

```
fn sample_function() {  
    println!("called `sample_function()`");  
}
```

4. Declare a module named `cool` using the `mod` keyword and create a function named `sample_function` with public visibility. Then, print `called cool::sample_function()` in its scope:

```
// Ddefined a module names cool  
mod cool {  
    pub fn sample_function() {  
        println!("called `cool::sample_function()` \n");  
    }  
}
```

5. Create another module using the `mod` keyword named `sample_mod` and create a function item named `sample_function` by printing "called `sample_mod::sample_function()`":

```
mod sample_mod {  
    fn sample_function() {  
        println!("called `sample_mod::sample_function()` \n");  
    }  
}
```

6. Create a module named `cool` with the function item `sample_function` by marking its visibility as public using the `pub` keyword, and printing "called `sample_mod::cool::sample_function()`":

```

mod cool {
    pub fn sample_function() {
        println!("called `sample_mod::cool::sample_function()``\\n");
    }
}

```

7. Create another function inside the `cool` module named `indirect_call` by marking its visibility as public using the `pub` keyword, and printing

`"called `sample_mod::indirect_call()`` , that\n> ":`

```

pub fn indirect_call() {
    // Let's access all the sample_functions named
    `sample_function` from
    this scope!
    print!("called `sample_mod::indirect_call()`` , that \\n > ");
}

```

Call `sample_function` using the `self` and `super` keywords:

```

// The `self` keyword refers to the current module scope - in
this case
`samp le_mod`.
// Calling `self::sample_function()` and calling
`sample_function()` directly both give
// the same result, because they refer to the same
sample_function.
self::sample_function();
sample_function();

// We can also use `self` to access another module inside
`samp le_mod`:
self::cool::sample_function();

// The `super` keyword refers to the parent scope (outside the
`samp le_mod` module).
super::sample_function();

```

8. Create a block and call `root_sample_function`, which is bound to

`cool::sample_function`:

```

// This will bind to the `cool::sample_function` in the *crate*
scope.
// In this case the crate scope is the outermost scope.
{
    use cool::sample_function as root_sample_function;
    root_sample_function();
}
}
}

```

9. Define the `main` function and call the sample module's `indirect_call` function:

```
// Execution starts here
fn main() {
    // Calling the sample_mod module's item
    sample_mod::indirect_call();
}
```

Upon the correct setup of the preceding code, you should get the following screenshot as output when you compile and run the program:

```
[vigneshwers-MacBook-Air:chapter4 vigneshwerdhinakaran$ rustc sample_access.rs
[vigneshwers-MacBook-Air:chapter4 vigneshwerdhinakaran$ ./sample_access
called `sample_mod::indirect_call()`, that
> called `sample_mod::sample_function()`

called `sample_mod::sample_function()`

called `sample_mod::cool::sample_function()`

called `sample_function()`
called `cool::sample_function()`
```


How it works...

Using the `super` and `self` keywords, we can remove ambiguity when accessing items across modules. This can help us eliminate a lot of hardcoding of paths.

We started off by creating a function named `sample_function`. In all the functions, we print how the function should be called. Then, we created a module named `cool` with a public function named `sample_function`, which had the same name as that of the declared outside the scope of `cool`. Lastly, we created a module named `sample_mod` consisting of a private function named `sample_function` and a public nested module `cool`, and a public function named `sample_function` and publicly visible function `indirect_call`.

All of the action in this recipe happens in the `indirect_call` function, which we call from the `main` function by `sample_mod::indirect_call()`. When we start to execute the `indirect_call` function, it first has a print statement that prints how the function was called, and then proceeds ahead with calling `self::sample_function()`. The `self` keyword refers to the current module scope. In this case, it was `sample_mod`, and calling `sample_function()` or `self::sample_function()` would have given the same result as they referred to the same `sample_function`.

To access `sample_function` of other modules (which in this case is `cool`) inside the scope of `sample_mod`, we have to mention the call using the `self` keyword, which is `self::cool::sample_function()`. To call the items/units outside the scope of the `sample_mod` module, we use `super`, which basically helps in calling the items outside the scope of the current module. Here, we called `sample_function` using the `super` keyword, which fetched the function that could be accessed by any units of the code. We achieved this by calling `super::sample_function()`. Next, we created a block in which we had the code chunk `use cool::sample_function as root_sample_function`, which used the `use` keyword to call `sample_function` of the `cool` module outside the scope and bind the path to `root_sample_function`.

Creating a file hierarchy

This recipe discusses how to create a file structure for complex and bigger code bases so that it would be easier for the developer to manage application feature development. We will learn about the rules enforced by the Rust compiler to create a file hierarchy successfully so that the developer can utilize and get the same flexibility while using modules' units.

Getting ready

We will require the Rust compiler and any text editor to code. Also, create a file named `sample_module.rs` in the project workspace.

How to do it...

1. Create a file named `sample_split.rs` and a folder named `sample_module` in the project workspace:

```
| touch sample_split.rs && mkdir sample_module
```

2. Create the `mod.rs`, `nested_mod.rs`, and `sample_private.rs` files inside the `sample_module` folder:

```
| cd sample_module && touch mod.rs nested_mod.rs  
| sample_private.rs
```

We should get a folder structure, as shown in the following screenshot:

```
viki@Vigneshwer:~/rust_cookbook/chapter4/sample_module$ tree .  
.  
|-- mod.rs  
|-- nested_mod.rs  
`-- sample_private.rs  
  
0 directories, 3 files
```

3. Write the code header with the details of the code in `sample_split.rs`:

```
| --- #####  
| --- Task: To create a sample file structure  
| --- Author: Vigneshwer.D  
| --- Version: 1.0.0  
| --- Date: 4 March 17  
| --- #####
```

4. Create a folder named `sample_module` containing the content of the `sample_mod module` in `sample_split.rs`:

```
| // Using the contents of sample_module  
| mod sample_module;
```

5. Declare a local function named `sample_function` that will print "called `sample_function()`", which will help in understanding the scope:

```
| // Defining a local sample_function  
| fn sample_function() {  
|     println!("called `sample_function()`");  
| }
```

6. Define the `main` function, where we can call various items that will help in understanding the code workflow in the modules:

```

// Execution starts here
fn main() {
    sample_module::sample_function();
    sample_function();
    sample_module::indirect_access();
    sample_module::nested_mod::sample_function();
}

```

7. Write the code header with the details of the code in `sample_module/mod.rs`:

```

///-- ######
///-- Task: To create a sample file structure
///-- Author: Vigneshwer.D
///-- Version: 1.0.0
///-- Date: 4 March 17
///-- #####

```

8. Declare the different modules that are `sample_private` and the publicly visible module `nested_mod` from the files inside the `sample_module` folder.



`mod.rs` is an essential Rust script file inside the module folder that helps the compiler understand the different contents of the module:

```

// Similarly `mod sample_private` and `mod nested_mod` will
locate the
`nested_mod.rs`
// and `sample_private.rs` files and insert them here under
their
respective
// modules
mod sample_private;
pub mod nested_mod;

```

9. Declare a public function named `sample_function` and print the called

``sample_module::sample_function()``:

```

pub fn sample_function() {
    println!("called `sample_module::sample_function()`");
}

```

10. Define a function named `private_function` and print the "called

``sample_module::private_function()``:

```

fn private_function() {
    println!("called `sample_module::private_function()`");
}

```

11. Declare a function named `indirect_access`, which calls `private_function` inside its scope:

```

pub fn indirect_access() {
    println!("called `sample_module::indirect_access()`, that \n");
}

```

```
|     private_function();  
| }
```

12. Write the code header with the details of the code in `sample_module/nested_mod.rs`:

```
| //--- #####  
| //--- Task: Nested module  
| //--- Author: Vigneshwer.D  
| //--- Version: 1.0.0  
| //--- Date: 4 March 17  
| //--- #####
```

13. Declare the items of the `nested_mod` module in this script. We start with defining the publicly visible `sample_function` and print "called"

```
`sample_module::nested::sample_function() `":
```

```
| // sample_mod/nested.rs  
| pub fn sample_function() {  
|     println!("called  
|         `sample_module::nested::sample_function() `");  
| }
```

14. Define a private function named `private_function` and print "called"

```
`sample_module::nested::private_function() `":
```

```
| #[allow(dead_code)]  
| fn private_function() {  
|     println!("called  
|         `sample_module::nested::private_function() `");  
| }
```

15. Write the code header with the details of the code in

```
sample_module/sample_private.rs:
```

```
| //--- #####  
| //--- Task: Inaccessible script  
| //--- Author: Vigneshwer.D  
| //--- Version: 1.0.0  
| //--- Date: 4 March 17  
| //--- #####
```

16. Define a publicly visible function named `public_function` inside the script and print "called `sample_module::sample_private::public_function() `":

```
| #[allow(dead_code)]  
| pub fn public_function() {  
|     println!("called  
|         `sample_module::sample_private::public_function() `");  
| }
```

Upon the correct setup of the preceding code, you should get the following screenshot as output when you compile and run the program:

```
[vigneshwers-MacBook-Air:chapter4 vigneshwerdhinakaran$ rustc sample_split.rs
[vigneshwers-MacBook-Air:chapter4 vigneshwerdhinakaran$ ./sample_split
called `sample_module::sample_function()`
called `sample_function()`
called `sample_module::indirect_access()`, that
> called `sample_module::private_function()`
called `sample_module::nested::sample_function()`
vigneshwers-MacBook-Air:chapter4 vigneshwerdhinakaran$
```


How it works...

It is not possible to have all the modules inside the same script for a large application, so we definitely need to follow a file hierarchy to maintain different units. We also need to learn the mapping to create a **file hierarchy**.

In the preceding recipe, we created the following files, which are `rs`, and a folder named `sample_module`, which has `mod.rs`, `nested_mod.rs`, and `sample_private.rs`.

The `mod.rs` is a mandatory file inside the directory where we mention other modules that the `sample_split.rs` function would use in the file. The `sample_module` uses the private module `sample_private` and public module `nested_mod`, which are declared at the top of the file. We also created two public functions `sample_function` and `indirect_access`, which call the private function named `private_function`. These items are part of `sample_module` and can be directly called by `sample_module` in the `sample_split.rs` file.

In the `nested_mod.rs` file, we created a public function named `sample_function`, which can be called, and a private function named `private_function`, which cannot be called. Because `sample_module` is a private element of `nested_mod` and, similarly, the `sample_private.rs` file, we have a public function named `public_function` that cannot be accessed, as the `sample_private` module is private.

In `sample_split.rs`, which is outside the `sample_module` directory, we used this script as the gateway to call the modules that were inside the `sample_module` folder. We started off by calling `mod sample_module`, which is the module name to call the contents of the directory. We also created a function named `sample_function`, local to `sample_split.rs`, for understanding the purpose. Then, in the `main` function, we called all the units.

First, we called `sample_module::sample_function()`, which is the element of the `sample_function` module itself. The working of `sample_module::indirect_access()`, which would call the private item of `sample_module`, would be similar. To call

the `nested_mod` public element, we called `sample_module::nested_mod::sample_function()`, which has the same syntax as that of the nested module.

Building libraries in Rust

In this recipe, you will learn how to build libraries that will contain the functional units of the Rust application, and the way we can compile the application in a library format so that we can access it externally from other programs.

Getting ready

We will require the Rust compiler and any text editor to code. Also, create a file named `sample_module.rs` in the project workspace.

How to do it...

1. Create a file named `sample_lib.rs` in the project workspace
2. Write the code header with the details of the code:

```
|    //--- #####  
|    //--- Task: To create a sample library in rust  
|    //--- Author: Vigneshwer.D  
|    //--- Version: 1.0.0  
|    //--- Date: 4 March 17  
|    //--- #####
```

3. Define a public visible function named `public_function` and print "called `sample_lib`public_function()`".

```
| pub fn public_function() {  
|     println!("called sample_lib`public_function()");  
| }
```

4. Define a private function named `private_function` and print "called `sample_lib`private_function()`".

```
| fn private_function() {  
|     println!("called sample_lib`private_function()");  
| }
```

5. Define another public function named `indirect_access` that will call `private_function`, declared in the preceding step, in its scope:

```
| pub fn indirect_access() {  
|     print!("called sample_lib`indirect_access()", that \n > );  
|     private_function();  
| }
```

Once the preceding code is set up, compile and run the project by the following command:

```
| rustc --crate-type=lib sample_lib.rs
```

We should get the following screenshot as output:

```
[vigneshwers-MacBook-Air:chapter4 vigneshwerdhinakaran$ rustc --crate-type=lib sample_lib.rs  
[vigneshwers-MacBook-Air:chapter4 vigneshwerdhinakaran$ ls lib*  
libsample_lib.rlib
```


How it works...

In this recipe, we created a sample library and made a `.rlib` extension package of the code. This will help us call the units of the library in other crates or Rust programs.

We created two public functions named `public_function` and `indirect_access` and a private function named `private_function`. We called `private_function` from `indirect_access`, which is a way through which we can call it outside the scope. Then, we created the library using the Rust compiler and passed a few command-line arguments to it, which tells the compiler to create the library format of the script.

While compiling the program, we ran `rustc --crate-type=lib sample_lib.rs`, which created a package named `libsample_lib.rlib` in the same directory. This file can be externally used at other crates. Alternatively, we can use Cargo to ship libraries by adding the `[lib]` tag in the `Cargo.toml` file.



Usually, libraries get prefixed with `lib`, and by default, they get the name of their crate file. But this default name can be overridden using the `crate_name` attribute while creating the library using `rustc`.

Calling external crates

In this recipe, you will learn how to use the external module units, libraries, or crates created from another Rust project. During the process, you will understand some basic syntax that allows external crate resources to be utilized as modules in the code, as well as the `extern crate` keyword, which provides a smooth way to call external crate resources.

Getting ready

We will require the Rust compiler and any text editor to code. Also, create a file named `sample_module.rs` in the project workspace.

How to do it...

1. Create a file named `sample_exec.rs` in the project workspace
2. Write the code header with the details of the code:

```
| --- #####  
| --- Task: To create a sample executor of sample_lib in rust  
| --- Author: Vigneshwer.D  
| --- Version: 1.0.0  
| --- Date: 4 March 17  
| --- #####
```

3. The previous script creates `libsample_lib.rlib`, which uses the `extern crate` keyword we are calling in this particular script:

```
| // Imports all items under sample_lib  
| extern crate sample_lib;
```

4. Declare the `main` function that calls all the public items of `sample_lib`, which are `public_function` and `indirect_access`:

```
| fn main() {  
|     // Calling public_function  
|     sample_lib::public_function();  
|     // Calling indirect_access to private_function  
|     sample_lib::indirect_access();  
| }
```

Once the preceding code is setup, compile and run the project by the following command:

```
| rustc sample_exec.rs --extern sample_lib=libsample_lib.rlib
```

We should get the following screenshot as output:

```
vigneshwers-MacBook-Air:chapter4 vigneshwerdhinakaran$ rustc sample_exec.rs --extern sample_lib=libsample_lib.rlib  
vigneshwers-MacBook-Air:chapter4 vigneshwerdhinakaran$ ./sample_exec  
called sample_lib 'public_function'  
called sample_lib 'indirect_access()', that  
> called sample_lib 'private_function'
```


How it works...

The aim of this recipe is to reuse the units created in the `libsample_lib.rlib` library in the code to link a crate to a new Rust script. We have to use the `extern crate` declaration to call the units. This will not only link the library, but also import all its items/units under the same module name as the library name, which in our case is `sample_lib`.



The visibility rules that apply to modules also apply to libraries.

Our first statement is `extern crate sample_lib`, which imports all the units. Now, we can call the units of the external library the way we call module items. In the `main` function, we call the units by `sample_lib::public_function()`; this would call `public_function` of `sample_lib` and `sample_lib::indirect_access()` would call `private_function` of `sample_lib`.

Deep Dive into Parallelism

In this chapter, we will cover the following recipes:

- Creating a thread in Rust
- Spawning multiple threads
- Holding threads in a vector
- Sharing data between threads using channels
- Implementing safe mutable access
- Creating child processes
- Waiting for a child process
- Making sequential code parallel

Introduction

Concurrency and parallelism are very important topics for creating a high-performance application that can completely utilize system resources, especially considering the fact that hardware is getting better with its offering of multiple cores.

Rust is a great programming language for performing parallel operations in your application. It ensures memory safety and freedom from data races, which is one of the major reasons for the list of various concurrency bugs. Rust utilizes the standard system APIs to perform concurrency operations.

Creating a thread in Rust

Rust's standard library provides various functionality for spawning threads, which allow the developer to develop and run Rust code in parallel. In this recipe, we will learn how to use `std::thread` for spawning multiple threads.

You will learn how to create a new thread from the parent, pass a value to the newly created child thread, and retrieve the value.

Getting ready

We will require the Rust compiler and any text editor for developing the Rust code snippet.

How to do it...

The following steps will walk you through this implementation:

1. Create a file named `sample_move.rs` and open it in your text editor.
2. Write the code header with the relevant information:

```
| //-- #####  
| //-- Task: Passing values to a thread in rust  
| //-- Author: Vigneshwer.D  
| //-- Version: 1.0.0  
| //-- Date: 19 March 17  
| //-- #####
```

3. Call the standard `thread` library using the `use` keyword:

```
| use std::thread;
```

4. Define the `main` function and declare two variables: `x` and `handle`. Assign `x` with the integer value `1` and assign `handle` to the new `thread`, using the `thread::spawn` command. Here's the code for this:

```
| fn main() {  
|     let x = 1;    let handle = thread::spawn(move || { (x) });  
|  
|     println!("{}:?", handle.join().unwrap());  
| }
```

You will get the following screenshot as an output upon successful execution of the code:

```
viki@Vigneshwer:~/rust_cookbook/chapter5/code$ rustc -A warnings sample_move.rs  
viki@Vigneshwer:~/rust_cookbook/chapter5/code$ ./sample_move  
1
```


How it works...

In this recipe, we use Rust's standard library, namely `std::thread`, for creating a `thread` that would allow your Rust code to run in parallel.

We created a variable named `x` using the `let` keyword and assigned it the value `1`, which we passed to the `thread` created by the `thread::spawn()` method. This method accepts a closure and it will be executed as a different `thread`; the result that it returns is collected in the `handle` variable of the main or parent `thread`, which is the originator of the `child` thread. The parent or main `thread` waits until the `child thread` completes the task, and by assigning it to a variable, we collect the information from the `child thread` in the `handle` variable.

As closures have the ability to capture variables from their environment, we brought the data from the `child thread` to the parent `thread`, but we have to do this carefully using the `move` closure. If you don't use `move`, you will get a compile-time error as, by default closures capture variables by reference, and we only have the reference to `x`. This is a problem of dangling pointers. The `move` closure prevents this by moving the variable from other environments to themselves.

At last, we called the `join()` and `unwrap()` methods to print the result from the `child thread`.

Spawning multiple threads

The aim of this recipe is to spawn multiple threads in Rust and perform simple actions that will help you understand how threads are generated in Rust. We will extensively use the standard `thread` module with the `move` closure, which we learned in the previous recipe.

Getting ready

We will require the Rust compiler and any text editor for developing the Rust code snippet.

How to do it...

The following steps will walk you through this implementation:

1. Create a file named `sample_multiple_threads.rs` and open it in your text editor.
2. Write the code header with the relevant information:

```
|    //--- #####  
|    //--- Task: Passing values to a thread in rust  
|    //--- Author: Vigneshwer.D  
|    //--- Version: 1.0.0  
|    //--- Date: 19 March 17  
|    //--- #####
```

3. Call the standard `thread` library using the `use` keyword:

```
| use std::thread;
```

4. Define the `main` function and declare two variables: `x` and `handle`. Assign `x` as an integer value of `1` and assign `handle` to the new `thread`, using the `thread::spawn` command. Here's the code for this:

```
| fn main() {  
|     thread::spawn(move || {  
|         println!("Hello from spawned thread");  
|     });  
  
|     let join_handle = thread::spawn(move || {  
|         println!("Hello from second spawned thread");  
|         17  
|     });  
  
|     println!("Hello from the main thread");  
  
|     match join_handle.join() {  
|         Ok(x) => println!("Second spawned thread returned {}", x),  
|         Err(_) => println!("Second spawned thread panicked")  
|     }  
| }
```

We will get the following screenshot as output upon successful execution of the code:

```
viki@Vigneshwer:~/rust_cookbook/chapter5/code$ rustc -A warnings sample_multiple_threads.rs
viki@Vigneshwer:~/rust_cookbook/chapter5/code$ ./sample_multiple_threads
Hello from the main thread
Hello from second spawned thread
Hello from spawned thread
Second spawned thread returned 17
viki@Vigneshwer:~/rust_cookbook/chapter5/code$ ./sample_multiple_threads
Hello from the main thread
Hello from spawned thread
Hello from second spawned thread
Second spawned thread returned 17
```


How it works...

In this recipe, we created two threads using the `spawn` module of the `thread` crate. The first `thread` just prints a statement, but the second `thread`, apart from printing, also passes and returns a value to the main `thread`. The main `thread` is the user-created Rust process that creates the other two threads.

The most important point to learn from this recipe is that the main `thread` will not wait for the spawned `thread` to complete, which means that the next `println!` macro won't be executed before the program exits. To ensure that the program waits for the threads to finish, we called the `join()` module on the `join_handle` variable of the `thread`.

We even send a value ¹⁷ to a different `thread` through `join_handle`, as we sent to the second `thread` in this case. In the last few lines of the code, we had the `match` statement, where we check whether the value `x` has returned from the second `thread` using `join_handle .join()`.



The preceding three `println!` statements can be observed in any order.

The reason why these statements could be observed in a different order in every execution is that they are scheduled by the OS.

Holding threads in a vector

We are going to create a train of 10 threads in this recipe and declare a vector where we will hold all the `thread` handles. We will join the main `thread` later to ensure the handles are executed and return the value that we would send while spawning the `thread`. We will extensively use the concepts learned from the previous two recipes.

Getting ready

We will require the Rust compiler and any text editor for developing the Rust code snippet.

How to do it...

The following steps will walk you through this implementation:

1. Create a file named `sample_thread_expt.rs` and open it in your text editor.
2. Write the code header with the relevant information:

```
///- #####  
///- Task: Spawning 10 threads in rust  
///- Author: Vigneshwer.D  
///- Version: 1.0.0  
///- Date: 19 March 17  
///- #####
```

3. Call the standard `thread` library using the `use` keyword and create a `static` variable called `NO_THREADS` using the 32-bit integer value `10`:

```
// Using the standard thread crate  
use std::thread;  
  
// static value NO_THREADS  
static NO_THREADS: i32 = 10;
```

4. Define the `main` function and declare an empty `thread_holder` vector. Then create the corresponding loops for pushing the threads spawned to the vector with the iterator value `i` and return them later:

```
// Main thread starts here  
fn main() {  
    // Make a mutable vector named thread_holder to hold the  
    // threads spawned  
    let mut thread_holder = vec![];  
  
    for i in 0..NO_THREADS {  
        // Spin up another thread  
        thread_holder.push(thread::spawn(move || {  
            println!("Thread number is {}", i);  
            i  
        }));  
    }  
  
    println!("*****");  
  
    for thread_elements in thread_holder {  
        // Wait for the thread to finish. Returns a result.  
        println!("Thread returned {:?}",  
            thread_elements.join().unwrap());  
    }  
}
```

We will get the following screenshot as output upon successful execution of the code:

```
viki@Vigneshwer:~/rust_cookbook/chapter5/code$ rustc -A warnings sample_thread_expt.rs
viki@Vigneshwer:~/rust_cookbook/chapter5/code$ ./sample_thread_expt
Thread number is 1
Thread number is 2
Thread number is 0
Thread number is 3
Thread number is 4
Thread number is 5
Thread number is 6
Thread number is 7
*****
Thread returned 0
Thread returned 1
Thread returned 2
Thread returned 3
Thread returned 4
Thread returned 5
Thread returned 6
Thread number is 9
Thread returned 7
Thread number is 8
Thread returned 8
Thread returned 9
viki@Vigneshwer:~/rust_cookbook/chapter5/code$ ./sample_thread_expt
Thread number is 0
Thread number is 2
Thread number is 3
Thread number is 1
Thread number is 4
Thread number is 5
Thread number is 6
Thread number is 7
Thread number is 8
*****
Thread number is 9
Thread returned 0
Thread returned 1
Thread returned 2
Thread returned 3
Thread returned 4
Thread returned 5
Thread returned 6
Thread returned 7
Thread returned 8
Thread returned 9
```


How it works...

We declared a static variable named `NO_THREADS` to which we assigned the 32-bit integer value `10`; we also created an empty mutable vector named `thread_holder`.

Using a `for` loop, we iterated over the value, starting from `0` to the upper limit value of the static variable, that is, `NO_THREADS`. Inside the `for` loop, we pushed the spawned threads to the `thread_holder` vector using the `push` functionality. While creating the `thread`, we used the `move` closure and sent the iterator value `i` to the newly spawned `thread`.

Once all the thread-spawning commands are completed by the compiler, we start to iterate over the `thread_holder` vector elements using a `for` loop. In this case, the iterator variable was `thread_elements`; we called the `join` and `unwrap` function to the `thread` through this variable. With this, we ensured that all the threads are completed and they return to the main `thread`, where we print the value that was sent earlier to the `thread` when it was spawned.

Since all the threads are scheduled by the OS, we can't predict the order in which threads will be spawned and the values returned.

Sharing data between threads using channels

The idea is to send a piece of information or data of the type `T` between threads via a channel. Here, `T` implements the `Send` trait, which indicates that variables or resources of the type `T` have the ability to transfer their ownership safely between threads. This particular feature of Rust helps in safe sharing of data between threads. This helps achieve safer concurrency and data-race freedom. The catch here is that the type `T`, which we want to send across the threads via the channel, must support and implement the `Send` trait. The second important trait is `Sync`. When `T` implements `Sync`, it means that something of this type has no possibility of introducing memory unsafety when used from multiple threads concurrently through shared references. These two traits allow you to use the Rust type system for making concurrent code.

Getting ready

We will require the Rust compiler and any text editor for developing the Rust code snippet.

How to do it...

The following steps will walk you through this implementation:

1. Create a file named `sample_channel.rs` and open it in your text editor.
2. Write the code header with the relevant information:

```
//-- #####  
//-- Task: Using channels to perform safe pass of data between  
threads  
//-- Author: Vigneshwer.D  
//-- Version: 1.0.0  
//-- Date: 19 March 17  
//-- #####
```

3. Call the standard `thread` library using the `use` keyword:

```
// Using standard libraries  
use std::sync::mpsc::{Sender, Receiver};  
use std::sync::mpsc;  
use std::thread;
```

4. Create a `static` variable called `NO_THREADS` with a 32-bit integer value of 3:

```
// Declaring number of threads  
static NO_THREADS: i32 = 3;
```

5. Define the `main` function and declare the `tx` and `rx` endpoints of the channel:

```
// Main thread starts  
fn main() {  
    // Creating endpoints of the channel  
    let (tx, rx): (Sender<i32>, Receiver<i32>) = mpsc::channel();
```

6. Create a `for` loop to create threads and send them across the channel:

```
for thread_no in 0..NO_THREADS {  
    // Closing the Sender  
    let thread_tx = tx.clone();  
  
    // Sending threads via the channel  
    thread::spawn(move || {  
        // thread sends the message to the channel  
        thread_tx.send(thread_no).unwrap();  
        println!("thread {} finished", thread_id);
```

```
|     });
| }
```

7. Similarly, create another `for` loop to iterate and collect all the values passed to the channel:

```
// Collecting all the threads
let mut thread_holder = Vec::with_capacity(NO_THREADS as
    usize);
for i in 0..NO_THREADS {
    // Get the message from channel
    thread_holder.push(rx.recv());
}

// Print the execution order
println!("{}:", thread_holder);
}
```

You will get the following output upon successful execution of the code:

```
viki@vigneshwer:~/rust_cookbook/chapter5/code$ rustc -A warnings sample_channel.rs
viki@vigneshwer:~/rust_cookbook/chapter5/code$ ./sample_channel
thread 0 finished
thread 2 finished
thread 1 finished
[Ok(0), Ok(2), Ok(1)]
viki@vigneshwer:~/rust_cookbook/chapter5/code$ ./sample_channel
thread 0 finished
thread 2 finished
thread 1 finished
[Ok(0), Ok(2), Ok(1)]
viki@vigneshwer:~/rust_cookbook/chapter5/code$ ./sample_channel
thread 0 finished
thread 2 finished
thread 1 finished
[Ok(0), Ok(1), Ok(2)]
viki@vigneshwer:~/rust_cookbook/chapter5/code$ ./sample_channel
thread 2 finished
thread 0 finished
thread 1 finished
[Ok(2), Ok(0), Ok(1)]
```


How it works...

In this recipe, we discussed how to use the channel feature of Rust to send data of the type `T`, which implements the traits required for safely sending data across threads.

First, to start off with developing channels, we used the `mpsc::channel()` method to create a new channel; post this, we sent simple data, such as the `thread_id`, from the endpoints. The endpoints in our case were `tx` and `rx`, which were the transmitter and receiver endpoints of the channel. The channel now had two endpoints, namely `Sender<T>` and `Receiver<T>`, where `T` was the type of the message that had to be transferred.

In the first `for` loop, where our focus was on sending the data to the channel, we iterated with a variable named `thread_id` from `0` to the static `NO_THREADS` variable value. The sender endpoint is copied by the `clone` method and assigned to `thread_id`. Each `thread` sends its `thread_no` via the channel by spawning new threads in which the `thread_id` value is passed to the `send(data).unwrap()` methods. The `thread` takes ownership of the `thread_tx` value. Each `thread` queues a message in the channel, sending a non-blocking operation, and continues immediately after sending the message.

In the second `for` loop, all the messages are collected from the channel. We declare a vector named `thread_holder` with the capacity of the number of threads spawned, which is a prefixed static value called `NO_THREADS`. The `recv` method of `rx` collects the messages from the channel, and `recv` blocks the current `thread` if there are no messages available. All these messages are pushed to the `thread_holder` vector using the `push` method of the vector. In the last `println` statement, we showed the order in which the messages were sent by printing the `thread_holder` vector.

Implementing safe mutable access

We want to ensure safe and mutable access of data, which will allow multiple threads to access the resource without having data races. The ownership model of Rust enables this functionality. You will learn about locks that will help you keep track of your data when you have multiple threads making modifications to it. We have the atomic reference count `Arc<T>` in Rust, which at runtime will keep track of the count and allow the developer to share the ownership of the data across threads.

In this recipe, we will use `mutex<T>`. This allows us to safely mutate a shared data value across multiple threads. For example, we have data where `mutex` will ensure only one `thread` would be able to mutate the value inside it at a time.

Getting ready

We will require the Rust compiler and any text editor for developing the Rust code snippet.

How to do it...

The following steps will walk you through this recipe:

1. Create a file named `sample_lock.rs` and open it in your text editor.
2. Write the code header with the relevant information:

```
///- #####  
///- Task: Safe Mutable access across threads for preventing  
data races  
///- Author: Vigneshwer.D  
///- Version: 1.0.0  
///- Date: 19 March 17  
///- #####
```

3. Call the standard libraries:

```
// Call the standard library  
use std::sync::{Arc, Mutex};  
use std::thread;  
use std::time::Duration;
```

4. Define the `main` function and declare the `data` variable with an `Arc` type data with `Mutex`:

```
// Main thread  
fn main() {  
    // Declaring a Arc type data  
    let data = Arc::new(Mutex::new(vec![1, 2, 3]));
```

5. Spawn multiple threads using the `for` loop, which will mutate the data using `lock`:

```
// Creating 3 threads and implementing lock  
for i in 0..3 {  
    let data = data.clone();  
    thread::spawn(move || {  
        let mut data = data.lock().unwrap();  
        data[0] += i;  
        println!("Thread id :{:?}", i);  
        println!("Data value :{:?}", data[0]);  
    });  
}  
  
thread::sleep(Duration::from_millis(10));
```

You will get the following output upon successful execution of the code:

```
viki@Vigneshwer:~/rust_cookbook/chapter5/code$ rustc -A warnings sample_lock.rs
viki@Vigneshwer:~/rust_cookbook/chapter5/code$ ./sample_lock
Thread id : 0
Data value :1
Thread id : 2
Data value :3
Thread id : 1
Data value :4
viki@Vigneshwer:~/rust_cookbook/chapter5/code$ ./sample_lock
Thread id : 0
Data value :1
Thread id : 1
Data value :2
Thread id : 2
Data value :4
viki@Vigneshwer:~/rust_cookbook/chapter5/code$ ./sample_lock
Thread id : 1
Data value :2
Thread id : 2
Data value :4
viki@Vigneshwer:~/rust_cookbook/chapter5/code$ ./sample_lock
Thread id : 0
Data value :1
Thread id : 2
Data value :3
Thread id : 1
Data value :4
```


How it works...

We created a new variable `data` of the `Mutex` type, which we implemented from the standard sync library. Using the `for` loop, we planned to spawn three threads. Inside the scope of the `for` loop, we cloned `data` to the same variable name: `data`. Next, using the standard threads library, we spawned three threads where we locked the `Mutex`.

The `Mutex` (short form of mutual exclusion) allows one `thread` to access a value at a time. If you wish to access the value, you have to use the `lock()` method on the type. This will lock the `Mutex`, and no other `thread` will be able to unlock it; therefore, no one will be able to modify the data. If a `thread` attempts to lock a `mutex` that is already locked, it will wait until the other `thread` releases the lock.



Note that the value of `i` is copied only to the closure and not shared among the threads.

The releasing of the lock is automatic as when the variable goes out of scope, it is automatically released, which makes it available to other threads.

Creating child processes

In this recipe, we will call a `child` process from the Rust code to the outside world and record its output values. A `child` process is created via the `Command` struct, which configures the spawning process. The `child` process, in general, is any other tool or application that you would run or start using specific shell commands. In Rust, we spawn these `child` processes from the main process, which is the main Rust application, and control them according to the need of the Rust application using the built-in methods that let us read errors, pass arguments, wait for the process to complete, parse the output, and more.

Getting ready

We will require the Rust compiler and any text editor for developing the Rust code snippet.

How to do it...

The following steps will walk you through the implementation:

1. Create a file named `sample_child_process.rs` and open it in your text editor.
2. Write the code header with the relevant information:

```
///-- #####  
///-- Task: To call a child process  
///-- Author: Vigneshwer.D  
///-- Version: 1.0.0  
///-- Date: 19 March 17  
///-- #####
```

3. Call the standard library:

```
// Call the standard library  
use std::process::Command;
```

4. Define the `main` function and declare the `output` variable, which is the `Command` implementation to execute the `child` process and get `std::output`:

```
// Main execution of the code  
fn main() {  
    // Command to be executed  
    let output = Command::new("rustc")  
        .arg("--version")  
        .output().unwrap_or_else(|e| {  
            panic!("failed to execute process: {}", e)  
        });
```

5. Print out the string value of `s` variable based on the `output` response using `if...else` statements:

```
// printing the output values  
if output.status.success() {  
    let s = String::from_utf8_lossy(&output.stdout);  
  
    print!("rustc succeeded and stdout was:n{}", s);  
}  
else {  
    let s = String::from_utf8_lossy(&output.stderr);  
  
    print!("rustc failed and stderr was:n{}", s);  
}
```

You will get the following output upon successful execution of the code:

```
viki@Vigneshwer:~/rust_cookbook/chapter5/code$ rustc -A warnings sample_child_process.rs
viki@Vigneshwer:~/rust_cookbook/chapter5/code$ ./sample_child_process
rustc succeeded and stdout was:
rustc 1.14.0 (e8a012324 2016-12-16)
```


How it works...

A `child` process is created via the `Command` struct, which is responsible for the spawning process. Here, we declared a variable `output` where we called the `new` method. This is the place where we entered the main command to run. Next, we had `arg`, which contained the options of a particular system command. The `output` response was responsible for getting the output and the other commands used for error handling.

The `struct child` basically has three fields: `pub stdin: Option<ChildStdin>`, `pub stdout: Option<ChildStdout>`, and `pub stderr: Option<ChildStderr>`. These fields handle represent functionalities such as standard input, output, and error, respectively.

In the `if...else` statement, we basically checked whether the `Command` had executed properly using the `status` module's `success` method, which returns `True` in case of success and `False` otherwise. In both the cases, we captured the error and output, which we printed by `&output.stdout` and `&output.stderr`.

Waiting for a child process

Often, we want the main thread to wait for the `child` process to complete before continuing with the main `thread` execution.

In this recipe, we will learn how to use the `wait` method to get the status of a `child` process.

Getting ready

We will require the Rust compiler and any text editor for developing the Rust code snippet.

How to do it...

The following steps will walk you through the recipe:

1. Create a file named `sample_wait.rs` and open it in your text editor.
2. Write the code header with the relevant information:

```
|    //--- #####  
|    //--- Task: Waiting for a child process  
|    //--- Author: Vigneshwer.D  
|    //--- Version: 1.0.0  
|    //--- Date: 19 March 17  
|    //--- #####
```

3. Call the standard library:

```
|    // Calling the standard libraries  
use std::process::Command;
```

4. Define the `main` function and create a `child` process using the `Command` struct:

```
|    // Main execution starts here  
fn main() {  
    // Creating a child process  
    let mut child =  
        Command::new("sleep").arg("5").spawn().unwrap();
```

5. Create a variable named `_result` and call the `wait` method, the last print statement, marking the end of the program:

```
|    // Waiting for the child process to complete  
let _result = child.wait().unwrap();  
  
    // printing the status of child process  
print!("Status of child process {} \n", _result);  
    // Marking the end of the main function  
    println!("reached end of main");  
}
```

You will get the following output upon successful execution of the code:

```
viki@Vigneshwer:~/rust_cookbook/chapter5/code$ rustc -A warnings sample_wait.rs
viki@Vigneshwer:~/rust_cookbook/chapter5/code$ ./sample_wait
Status if child process exit code: 0
Reached end of main
```


How it works...

In this recipe, we created a variable named `child` in which we called the `command` struct to create the `child` process, which is the `sleep` statement with an argument with `5`. This makes the program sleep or waits for five seconds.

Calling the `wait` function will make the parent or the main process wait until the `child` process has actually exited before continuing with the other statements of the main process. The mutable `child` `wait` function waits for the command to exit completely and returns the status that it exited with to the `_result` variable.

Last, we printed the status of the `child` process and marked the end of the main `thread`.

Making sequential code parallel

Here, you'll learn about `rayon`, an external crate in Rust, whose main aim is to make your sequential code parallel. The best part about `rayon` is that it guarantees you that the APIs of `rayon` will not introduce any concurrency bugs, such as data race. In this recipe, you will learn about rayon's parallel iterators, which execute iterative statements in parallel.

Getting ready

Follow these steps to download and set up a `rayon` crate in your project:

1. We will require the Rust compiler and any text editor for developing the Rust code `snippet.cargo`.
2. Create a new Cargo project named `sample_rayon` using:

```
| cargo new sample_rayon --bin
```

You will get the following screenshot as output:

```
viki@Vigneshwer:~/rust_cookbook/chapter5/code$ cargo new sample_rayon --bi
      Created binary (application) `sample_rayon` project
```

3. Enter the newly created `sample_rayon` project and check whether the structure is created properly:

```
| cd sample_rayon/ && tree .
```

You will get the following output:

```
viki@Vigneshwer:~/rust_cookbook/chapter5/code$ cd sample_rayon/
viki@Vigneshwer:~/rust_cookbook/chapter5/code/sample_rayon$ tree .
.
|-- Cargo.toml
|-- src
   '-- main.rs
1 directory, 2 files
```

4. Open the `Cargo.toml` file to download the `rayon` crate using:

```
| nano Cargo.toml
```

5. Enter the following values in the dependencies tag:

```
viki@Vigneshwer:~/rust_cookbook/chapter5/code/sample_rayon$ cat Cargo.toml
[package]
name = "sample_rayon"
version = "0.1.0"
authors = ["Vigneshwer.D <d.vigneshwer@gmail.com>"]

[dependencies]
rayon = "0.6.0"
```

6. Build the project again to download the `rayon` crate:

```
| cargo build
```

You will get the following screenshot as output:

```
viki@Vigneshwer:~/rust_cookbook/chapter5/code/sample_rayon$ cargo build
  Updating registry `https://github.com/rust-lang/crates.io-index'
  Downloading rayon v0.6.0
  Downloading rand v0.3.15
  Downloading libc v0.2.21
  Downloading num_cpus v1.3.0
  Downloading deque v0.3.1
    Compiling libc v0.2.21
    Compiling rand v0.3.15
    Compiling num_cpus v1.3.0
    Compiling deque v0.3.1
    Compiling rayon v0.6.0
    Compiling sample_rayon v0.1.0 (file:///home/viki/rust_cookbook/chapter5/code/sample_rayon)
      Finished debug [unoptimized + debuginfo] target(s) in 6.82 secs
viki@Vigneshwer:~/rust_cookbook/chapter5/code/sample_rayon$ ./target/debug/sample_rayon
Sum of squares of 10 is 100
```

7. Check the file structure to see the dependencies installed:

```
| ls && tree .
```

You will get the following screenshot as output:

```
viki@Vigneshwer:~/rust_cookbook/chapter5/code/sample_rayon$ ls && tree .
Cargo.lock  Cargo.toml  src  target
.
|-- Cargo.lock
|-- Cargo.toml
|-- src
|   '-- main.rs
`-- target
    '-- debug
        '-- build
        '-- deps
            '-- libdequeue-82214e5f75d78bdf.rlib
            '-- liblibc-1d4b292c3e055073.rlib
            '-- libnum_cpus-2c16f3104e5429bb.rlib
            '-- librand-8ea7d489d4a383a0.rlib
            '-- librayon-ecad47f1e1ddbdcc6.rlib
        '-- examples
        '-- native
        '-- sample_rayon

7 directories, 9 files
```


How to do it...

The following steps will walk you through the implementation:

1. Create a file named `sample_rayon.rs` and open it in your text editor.
2. Write the code header with the relevant information:

```
//-- #####  
//-- Task: Making sequential code parallel  
//-- Author: Vigneshwer.D  
//-- Version: 1.0.0  
//-- Date: 19 March 17  
//-- #####
```

3. Call the external library, named `rayon`, which we developed in the *Getting ready* section:

```
// Calling the rayon crate  
extern crate rayon;  
use rayon::prelude::*;


```

4. Define the `sum_of_squares` function and accept a variable `input` of the type `i32`:

```
// Sum of squares function  
fn sum_of_squares(input: &[i32]) -> i32 {  
    input.par_iter()  
    .map(|&i| i * i)  
    .sum()  
}
```

5. Define the `main` function where we will create `rand_val` and assign it the value `10`, which we will pass to the `sum_of_square` function:

```
// Main execution of code  
fn main() {  
    // Declaring a random variable of 10  
    let rand_val = 10;  
    // Calling the method to get sum_of_squares  
    let sum_sq = sum_of_squares(&[rand_val]);  
    // Printing the result  
    println!("Sum of squares of {} is {}", rand_val, sum_sq);  
}
```

You will get the following screenshot as output upon successful execution of the code:

```
viki@Vigneshwer:~/rust_cookbook/chapter5/code/sample_rayon$ ./target/debug/sample_rayon
Sum of squares of 10 is 100
```


How it works...

Since `rayon` is an external crate, we first downloaded it using the Cargo tool from the [crates.io](#) repo site. We used the latest version of `rayon`, which is `0.6.0`. `rayon` currently requires rustc 1.12.0 and is available in [crates.io](#). And note, `rayon` is an experimental crate.

`rayon` has an API called parallel iterators that lets us write iterator processes and execute them in parallel. We implemented the `sum_of_squares` function using parallel iterators. To use parallel iterators, we first imported the traits by calling `use rayon::prelude::*;` to our Rust module. We then called `par_iter` to get a parallel iterator, which is similar to a regular iterator. Parallel iterators work by first constructing a computation and then executing it; `input` is the parameter of the `sum_of_squares` function that returns an integer output. We performed the mathematical operation using the `map` and `sum` methods.

In the `main` function, we created a variable named `rand_var` and assigned it the value `10`, which we passed as an argument to the `sum_of_squares` function. The return value is stored in the `sum_sq` variable and printed.

Efficient Error Handling

In this chapter, we will cover the following recipes:

- Implementing panic
- Implementing Option
- Creating map combinator
- Creating and_then combinator
- Creating map for the Result type
- Implementing aliases
- Handling multiple errors
- Implementing early returns
- Implementing the try! macro
- Defining your own error types
- Implementing the boxing of errors

Introduction

Error handling is a fundamental part of all programming languages. It is the way in which a developer prepares for the worst conditions by noticing and managing errors due to which the application could fail. These error conditions can occur due to various reasons, such as wrong input provided at runtime and more. In this chapter, we will cover various methods using which we can efficiently handle errors in Rust. We'll also check out the standard library that helps avoid problematic situations and thus avoid a complete failure of the Rust application.

Implementing panic

Panic is the simplest error handling mechanism provided by Rust. It prints the error messages given to it, starts to unwind the task, and usually exits the program execution. In this recipe, we will explicitly call out a `panic` statement in the face of an undesired case.

Getting ready

We will require the Rust compiler and any text editor for developing the Rust code snippet.

How to do it...

Follow the ensuing steps to get through this recipe:

1. Create a file named `sample_panic.rs` and open it in your text editor.
2. Write the code header with the relevant information:

```
//-- #####  
//-- Task: Implementing panic  
//-- Author: Vigneshwer.D  
//-- Version: 1.0.0  
//-- Date: 26 March 17  
//-- #####
```

3. Create the `compare_stmt` function; it accepts a string input:

```
// function which checks if the strings are same or not  
fn compare_stmt(stmt: &str) {  
    // Check if the statements are same or not  
    if stmt == "Another book" {  
        panic!("Rust Cookbook is not selected!!!!");  
    }  
  
    println!("Statements is {}!!!!", stmt);  
}
```

4. Define the `main` function; it calls the `compare_stmt` function with different input:

```
// Execution starts here  
fn main() {  
    compare_stmt("Rust Cookbook");  
    compare_stmt("Another book");  
}
```

You will get the following output upon successful execution of the code:

```
viki@Vigneshwer:~/rust_cookbook/chapter6/code$ rustc -A warnings sample_panic.rs  
viki@Vigneshwer:~/rust_cookbook/chapter6/code$ ./sample_panic  
Statements is Rust Cookbook!!!!  
thread 'main' panicked at 'Rust Cookbook is not selected!!!!', sample_panic.rs:12  
note: Run with `RUST_BACKTRACE=1` for a backtrace.
```


How it works...

We have a function named `compare_stmt` that accepts an `str` variable as an argument and assigns it to a variable named `input` in its scope. It later checks whether the string value is `Another Book`. If it is, it calls the `panic!` function; otherwise, it prints the value that was passed. We passed two values from the main function: `Rust CookBook` and ``Another Book``.

When we run the preceding program, the first input will fail the `if` condition and will not invoke panic, so we get the print statement working. However, for the second input, which satisfies the `if` condition, panic is invoked and it returns `thread 'main' panicked at 'Rust Cookbook is not selected!!!!'`, `sample_panic.rs:12` and exits the program.

Implementing Option

Panic handles cases where there are identified instances of undesired input, but it does not handle zero input. For that, we have the `Option<T>` type, an `enum` from the standard library that you can use to handle cases where you have no input. In this recipe, you will learn the different ways in which you can use Options to handle zero case input.

Getting ready

We will require the Rust compiler and any text editor for developing the Rust code snippet.

How to do it...

Follow the ensuing steps to get through this recipe:

1. Create a file named `sample_option.rs` and open it in your text editor.
2. Write the code header with the relevant information:

```
//-- #####  
//-- Task: Implementing Option  
//-- Author: Vigneshwer.D  
//-- Version: 1.0.0  
//-- Date: 26 March 17  
//-- #####
```

3. Create the `compare_stmt_match` function; it accepts the `input` string of the `Option<&str>` type:

```
// All arguments are handled explicitly using `match`.  
fn compare_stmt_match(input: Option<&str>) {  
    // Specify a course of action for each case.  
    match input {  
        Some("Rust CookBook") => println!("Rust CookBook  
was selected"),  
        Some(inner) => println!("Rust CookBook not  
selected"),  
        None => println!("No input provided"),  
    }  
}
```

4. Similarly, create the `compare_stmt_unwrap` function; it also accepts the `input` string of the `Option<&str>` type:

```
// All arguments are handled implicitly using `unwrap`.  
fn compare_stmt_unwrap(input: Option<&str>) {  
    // `unwrap` returns a `panic` when it receives a  
    // `None` value  
    let inside_val = input.unwrap();  
    if inside_val == "Another Book" { panic!("Rust  
CookBook is not selected"); }  
  
    println!("I love {}s!!!!!", inside_val);  
}
```

5. Define the `main` function; it calls the two functions with different input:

```
// main execution starts here
fn main() {
    let Desired_Book = Some("Rust CookBook");
    let Another_Book = Some("Another Book");
    let Empty_value = None;

    compare_stmt_match(Desired_Book);
    compare_stmt_match(Another_Book);
    compare_stmt_match(Empty_value);

    println!("*****");
    let Rand_Book = Some("Random Book");
    let No_val = None;

    compare_stmt_unwrap(Rand_Book);
    compare_stmt_unwrap(No_val);
}
```

You will get the following output upon successful execution of the code:

```
viki@Vigneshwer:~/rust_cookbook/chapter6/code$ rustc -A warnings sample_option.rs
viki@Vigneshwer:~/rust_cookbook/chapter6/code$ ./sample_option
Rust CookBook was selected
Rust CookBook not selected
No input provided
*****
I love Random Books!!!!
thread 'main' panicked at 'called `Option::unwrap()` on a `None` value', ../../src/libcore/option.rs:323
note: Run with 'RUST_BACKTRACE=1' for a backtrace.
```


How it works...

In the preceding recipe, we knew we had to exit the program using `panic!` in the case of an undesired input, but the main problem we are trying to solve in this recipe is the way by which we can handle `None` input. We use the Rust standard library to address this problem. More specifically, we use an `enum` called `Option<T>` from the `std` library, which is used when there is no input:

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

It has two options, namely:

- `Some(T)`: This is an element of the type `T` that was sent
- `None`: This refers to the case where there was no input

We handle these cases in two ways: the explicit way of handling in which we use `match` and the implicit way in which we use `unwrap`. The implicit way of handling returns the inner element of either `enum` or `panic!`.

In the explicit way of handling, we declared three variables, namely `Desired_Book`, `Another_Book`, and `Empty_value` in the `main` function. We assigned them with book names, which were `Rust Cookbook`, `Another Book`, and `'None'`, respectively. Post this, we called the functions in the following manner:

- `compare_stmt_match(Desired_Book)`: This satisfies the `match` statement condition `Some("Rust CookBook")` to print `Rust CookBook was selected`
- `compare_stmt_match(Another_Book)`: This satisfies the `match` statement condition `Some(inner)` to print `"Rust CookBook not selected"`
- `compare_stmt_match(Empty_val)`: This satisfies the `match` statement condition `None` to print `No input provided`

In implicit handling, we created `Rand_Book` and `No_val` with the values `Some("Random Book")` and `None`, respectively. We call another function that uses `unwrap` to handle `Some(T)` and `None` values. The `compare_stmt_unwrap(Rand_Book)`

used `unwrap` to get `inside_val`, which successfully called the print statement; on the second function call `compare_stmt_unwrap(No_val)`, we got thread 'main' panicked at 'called `Option::unwrap()` on a `None` value',
.../src/libcore/option.rs:323. This was because `unwrap` returns a panic when we have `None` as the inner value.

Creating map combinator

We will learn about the map combinator in this recipe, which again is a combinator for handling `Option` types. The `Option` has an inbuilt map method for simple mapping of `Some(T)` to another valid type; it can also handle the mapping of none values. The `map` is a great way to explicitly handle `None` case input. It also simplifies the code as it can be used multiple times.

Combinators, in general, are high-order functions that apply only the functions and the combinators defined earlier to provide a result from their arguments. They are generally used to control the flow in a modular fashion in an application.

Getting ready

We will require the Rust compiler and any text editor for developing the Rust code snippet.

How to do it...

Follow these steps to get through this recipe:

1. Create a file named `sample_map.rs` and open it in your text editor.
2. Write the code header with the relevant information:

```
///-- #####  
///-- Task: Implementing map  
///-- Author: Vigneshwer.D  
///-- Version: 1.0.0  
///-- Date: 26 March 17  
///-- #####
```

3. Create a user-defined data type to use the program:

```
#[allow(dead_code)]  
  
#[derive(Debug)] enum Food { Apple, Carrot, Potato }  
  
#[derive(Debug)] struct Peeled(Food);  
#[derive(Debug)] struct Chopped(Food);  
#[derive(Debug)] struct Cooked(Food);
```

4. Define the `peel` function; it accepts the `Option<Food>` type input and returns `Option<Peeled>`:

```
fn peel(food: Option<Food>) -> Option<Peeled> {  
    match food {  
        Some(food) => Some(Peeled(food)),  
        None => None,  
    }  
}
```

5. Define the `chop` function; it accepts the `Option<Peeled>` type input and returns `Option<Chopped>`:

```
fn chop(peeled: Option<Peeled>) -> Option<Chopped> {  
    match peeled {  
        Some(Peeled(food)) => Some(Chopped(food)),  
        None => None,  
    }  
}
```

6. Define the `cook` function; it accepts the `Option<Chopped>` type input and returns `Option<Cooked>`:

```
| fn cook(chopped: Option<Chopped>) -> Option<Cooked> {
|     chopped.map(|Chopped(food)| Cooked(food))
| }
```

7. Define the `process` function; it accepts the `Option<Food>` type input and returns `Option<Cooked>`:

```
| fn process(food: Option<Food>) -> Option<Cooked> {
|     food.map(|f| Peeled(f))
|         .map(|Peeled(f)| Chopped(f))
|             .map(|Chopped(f)| Cooked(f))
| }
```

8. Define the `eat` function; it accepts the `Option<Cooked>` type input:

```
| fn eat(food: Option<Cooked>) {
|     match food {
|         Some(food) => println!("Mmm. I love {:?}", food),
|         None => println!("Oh no! It wasn't edible."),
|     }
| }
```

9. Define the `main` function where we can create the different types of input to understand the working of the map combinator:

```
| fn main() {
|     let apple = Some(Food::Apple);
|     let carrot = Some(Food::Carrot);
|     let potato = None;
|
|     let cooked_apple = cook(chop(peel(apple)));
|     let cooked_carrot = cook(chop(peel(carrot)));
|     let cooked_potato = process(potato);
|
|     eat(cooked_apple);
|     eat(cooked_carrot);
|     eat(cooked_potato);
| }
```

You will get the following output upon successful execution of the code:

```
viki@Vigneshwer:~/rust_cookbook/chapter6/code$ rustc -A warnings sample_map.rs
viki@Vigneshwer:~/rust_cookbook/chapter6/code$ ./sample_map
Mmm. I love Cooked(Apple)
Mmm. I love Cooked(Carrot)
Oh no! It wasn't edible.
```


How it works...

We first created an `enum` type named `Food` that had data elements, namely `Apple`, `Carrot`, and `Potato`. Then we created three `struct` with user-defined data types, namely `Peeled`, `Chopped`, and `Cooked` with `Food` as a data field. In the `main` function, we created three variables and assigned them to the values of the `option` data, where `apple` was valued `Food::Apple`, `carrot` as `Food::Carrot`, and `potato` as `None`.

Now let's check out how our function units react to different input:

- `peel`: This function takes in an `option` type that has a field `enum` type `Food` along with the data and returns an `Option` of the `struct` data type `Peeled`. Here we use the `match` function to change the type.
- `chop`: This function takes in the `option` type that has a field `enum` type `Peeled` along with the data and returns an `Option` of the `struct` data type `Chopped`. Here we use the `match` function to change the type.
- `cook`: This function takes in the `option` type that has a field `enum` type `Chopped` along with the data and returns an `Option` of the `struct` data type `Cooked`. Here we use the `map` function to change the type, where we place the input type between two pipe symbols that convert them into the desired form.
- `process`: Instead of having three functions to change types, we use `map` multiple times to convert `Option<Food>` into `Option<Cooked>` directly, where each `map` function successively converts the type to the desired form by this process we can `peel`, `chop`, and `cook` `food` type in a sequence by using multiple `map()`, thus it simplifies the code.
- `eat`: This function takes in the `Option<Cooked>` type as an input argument and checks it using a `match` statement. The first case `some(food)` would be true if a valid type exists for the `food` argument which is passed to the `match` statement, then it would print the value of the argument `food` in the place holder of the `print` statement else in the `None` case, it prints a default statement.

In the `main` function, we declared `cooked_apple` and assigned the return value of the `chop(peel(apple))` call. Since we didn't pass a `None` input, this was supposed to return the `Cooked(apple)` type of the data feed. Similarly, `cooked_carrot` had the value `Cooked(carrot)`; however, `cooked_potato`, for which we called the `process` function, returned `None`. Later, when we called the `eat` function, only the variable that had `Cooked` struct values got printed as `MM. I Love` statement and the variable that had `None` had `Oh No!` statement.

Creating and_then combinator

The problem with `map` is that it can get confusing when we have too many functions returning maps. This is because the result will be nested in the `Option<Option<T>>` format, and this gets complicated and confusing on multiple calls of the map combinator.

Rust provides another combinator, namely `and_then()`, which solves the preceding problem by only returning the result instead. It does it by flattening the chained results to a single type. In this recipe, you will learn how to use this combinator in detail.

Getting ready

We will require the Rust compiler and any text editor for developing the Rust code snippet.

How to do it...

Follow these steps to get through this recipe:

1. Create a file named `sample_and_then.rs` and open it in your text editor.
2. Write the code header with the relevant information:

```
|    //--- #####  
|    //--- Task: Implementing and_then  
|    //--- Author: Vigneshwer.D  
|    //--- Version: 1.0.0  
|    //--- Date: 26 March 17  
|    //--- #####
```

3. Create the `enum` types `Food` and `Day`:

```
|    #![allow(dead_code)]  
  
|    #[derive(Debug)] enum Food { CordonBleu, Steak, Sushi }  
|    #[derive(Debug)] enum Day { Monday, Tuesday, Wednesday }
```

4. Define a function named `have_ingredients` that will accept the `Food` type as an input argument and return `Option<Food>`:

```
|    fn have_ingredients(food: Food) -> Option<Food> {  
|        match food {  
|            Food::Sushi => None,  
|            _ => Some(food),  
|        }  
|    }
```

5. Define a function named `have_recipe` that will accept the `Food` type as an input argument and return `Option<Food>`:

```
|    fn have_recipe(food: Food) -> Option<Food> {  
|        match food {  
|            Food::CordonBleu => None,  
|            _ => Some(food),  
|        }  
|    }
```

6. Define a function named `cookable` that will accept the `Food` type as an input argument and return `Option<Food>`:

```
|    fn cookable(food: Food) -> Option<Food> {  
|        have_ingredients(food).and_then(have_recipe)
```

```
|     }
```

7. Define a function named `eat` that will accept the `Food` type as an input argument and return `Day`:

```
fn eat(food: Food, day: Day) {
    match cookable(food) {
        Some(food) => println!("Yay! On {:?}", we get to eat
        {:?}.", day, food),
        None => println!("Oh no. We don't get to eat on
        {:?}?", day),
    }
}
```

8. Define the `main` function; it will initialize and call all the functions:

```
fn main() {
    let (cordon_bleu, steak, sushi) = (Food::CordonBleu,
    Food::Steak, Food::Sushi);

    eat(cordon_bleu, Day::Monday);
    eat(steak, Day::Tuesday);
    eat(sushi, Day::Wednesday);
}
```

You will get the following output upon successful execution of the code:

```
viki@Vigneshwer:~/rust_cookbook/chapter6/code$ rustc -A warnings sample_and_then.rs
viki@Vigneshwer:~/rust_cookbook/chapter6/code$ ./sample_and_then
Oh no. We don't get to eat on Monday?
Yay! On Tuesday we get to eat Steak.
Oh no. We don't get to eat on Wednesday?
```


How it works...

We declared two `enum` types, namely `Food` and `Day`, where we had these elements: `CordonBleu`, `Steak`, and `Sushi` and `Monday`, `Tuesday`, and `Wednesday`.

Now let's see how our function units react to different input:

- `have_ingredients`: This function takes in the `enum` input `Food` and returns `Option<Food>`. It has a case in its `match` statement indicating whether `Food` has the value `Sushi` in that it returns `None` and for all other values, it returns the same `Food` value.
- `have_recipe`: This function takes in the `enum` input `Food` and returns `Option<Food>`. It has a case in its `match` statement indicating whether `Food` has the value `CordonBleu` in that it returns `None`; for all other values, it returns the same `Food` value.
- `cookable`: This function takes in the `enum` input `Food` and returns `Option<Food>`. Here, we use the `and_then` combinator to check the `have_ingredients` and `have_recipe` functions in order to confirm that the `Food` type will pass these cases.
- `eat`: This function takes the `enum` input `Food` and `Day` and sends the `Food` value to the `cookable` function. Here we have a `match` statement that prints the day and `Food` type in the case of `Some(Food)` from the `cookable` function.

We observe that for `cookable` to return a value, we need both the functions `have_ingredients` and `have_recipe` to return `Some(Food)`, which happens only in the case of `Steak`. In the `main` function, we called the `eat` function with all the values of `Food` and `Day`.

Creating map for the Result type

The `Result` type is similar to the `Option` type, but it offers more, as it also describes the possible error. This means we will have two outcomes: one where the desired element is found and the other where we may have found an error with an element. In this recipe, we will use the `map` method of `Result` to return a specific error.

Getting ready

We will require the Rust compiler and any text editor for developing the Rust code snippet.

How to do it...

Follow these steps to get through this recipe:

1. Create a file named `sample_map_result.rs` and open it in your text editor.
2. Write the code header with the relevant information:

```
|    //--- #####  
|    //--- Task: Implementing map for Result  
|    //--- Author: Vigneshwer.D  
|    //--- Version: 1.0.0  
|    //--- Date: 26 March 17  
|    //--- #####
```

3. Call the standard library:

```
|    use std::num::ParseIntError;
```

4. Create a function named `double_number` that will accept the `str` input and return a `Result<T>` type:

```
|    fn double_number(number_str: &str) -> Result<i32,  
|        ParseIntError> {  
|        match number_str.parse::<i32>() {  
|            Ok(n) => Ok(2 * n),  
|            Err(e) => Err(e),  
|        }  
|    }
```

5. Create a function named `double_number_map` that will accept the `str` input and return a `Result<T>` type:

```
|    fn double_number_map(number_str: &str) -> Result<i32,  
|        ParseIntError> {  
|        number_str.parse::<i32>().map(|n| 2 * n)  
|    }
```

6. Create a function named `print` that will accept a `Result<T>` type as input:

```
|    fn print(result: Result<i32, ParseIntError>) {  
|        match result {  
|            Ok(n) => println!("n is {}", n),  
|            Err(e) => println!("Error: {}", e),  
|        }  
|    }
```

7. Define the `main` function and declare different input for different functions:

```
fn main() {  
    // This still presents a reasonable answer.  
    let twenty = double_number("10");  
    print(twenty);  
  
    // The following now provides a much more helpful  
    // error message.  
    let tt = double_number_map("t");  
    print(tt);  
}
```

You will get the following output upon successful execution of the code:

```
viki@Vigneshwer:~/rust_cookbook/chapter6/code$ rustc -A warnings sample_map_result.rs  
viki@Vigneshwer:~/rust_cookbook/chapter6/code$ ./sample_map_result  
n is 20  
Error: invalid digit found in string
```


How it works...

Panicking gives us only an error message, which is not of great use for being more specific regarding the return type and the error. We have the `Result` type, which is similar to `Options`, but it can also be used to mention the error type. In this recipe, we used the `map` method of Rust to get the specific error type.

First, we called the standard library error type `ParseIntError`, which we used for returning error types as per the `Result` type.

Let's check out the different functional units in the code:

- `print`: This function takes in an input type of `Result<i32, ParseIntError>`, and based on its value, whether it is `Ok` or `Err`, prints the corresponding statements.
- `double_number_map`: This function takes in the `str` input and returns `Result<i32, ParseIntError>`. It parses the string to a value, and if it is a valid integer, we use the `map` function to multiply the input value by 2; else, we have the `Err` case.
- `double_number`: This function takes in the `str` input and returns `Result<i32, ParseIntError>`, where it has a `match` statement where it parses the string to a value. If it is a valid integer, it satisfies the `Ok` case and the value is multiplied by two if an `Err` case occurs.
- `print`: This function takes in `Result<i32, ParseIntError>`; using the `match` statement, we check whether we have an `Ok` or `Err` case for printing the corresponding statement.

In the `main` function, we had two variables, namely `twenty` and `tt`, assigned to `double_number("10")` and `double_number_map("t")`, respectively. When we called `double_number` function for `twenty`, it returned an integer value, but `double_number_map` returned an error for `t`. The print statement printed the final value of these `Result` type variables. In the case of passing a string to the `double_number` method that can't be parsed to integer will result in the

`ParseIntError` and a valid integer argument of string type will result in double its value for `double_number_map`.

Implementing aliases

We use aliases in order to reuse a specific type multiple times. Rust allows us to create aliases of the `Result` type and more types in order to reuse them across the program. At the module level, this is really helpful as we can identify similar kinds of bugs and errors from the units/items of the module.

Getting ready

We will require the Rust compiler and any text editor for developing the Rust code snippet.

How to do it...

Follow these steps to get through this recipe:

1. Create a file named `sample_aliases_result.rs` and open it in your text editor.
2. Write the code header with the relevant information:

```
|    //-- #####  
|    //-- Task: Implementing aliases  
|    //-- Author: Vigneshwer.D  
|    //-- Version: 1.0.0  
|    //-- Date: 26 March 17  
|    #####
```

3. Call the standard library:

```
|    use std::num::ParseIntError;
```

4. Define a generic alias named `AliasedResult<T>` for the `Result` type:

```
|    type AliasedResult<T> = Result<T, ParseIntError>;
```

5. Create a function named `double_number` that will accept the `str` input and return an `AliasedResult<i32>` type:

```
|    fn double_number(number_str: &str) -> AliasedResult<i32> {  
|        number_str.parse::<i32>().map(|n| 2 * n)  
|    }
```

6. Create a function named `print` that will accept an `AliasedResult<i32>` type as input:

```
|    fn print(result: AliasedResult<i32>) {  
|        match result {  
|            Ok(n) => println!("n is {}", n),  
|            Err(e) => println!("Error: {}", e),  
|        }  
|    }
```

7. Define the `main` function and call the different functions:

```
|    fn main() {  
|        print(double_number("10"));  
|        print(double_number("t"));  
|    }
```

You will get the following output upon successful execution of the code:

```
viki@Vigneshwer:~/rust_cookbook/chapter6/code$ rustc -A warnings sample_aliases_result
viki@Vigneshwer:~/rust_cookbook/chapter6/code$ ./sample_aliases_result
n is 20
Error: invalid digit found in string
```


How it works...

Aliases work in a fairly simple manner. Its main role is to ease the usage of a particular type and maintain different types across different modules.

In this recipe, we used the `type` keyword to create an alias for `Result<T, ParseIntError> as AliasedResult<T>`; we used this as the type of all the units of the code.

Let's go through the different functional units of the code:

- `double_number`: This function takes in the `str` input and returns `AliasedResult<T>`. It parses the string to a value, and if it is a valid integer, we use the `map` function to multiply the input value by 2; else, we have the `Err` case.
- `print`: This function takes in `AliasedResult<T>`, and using the `match` statement, we check whether we have an `Ok` or `Err` case for printing the corresponding statement.

In the `main` function, we called `print(double_number("10"))`, which printed the `Ok` case statements due to valid input, but `print(double_number("t"))` pushed back an error due to invalid input.

Handling multiple errors

In the previous recipes, we saw and developed error-handling units, where Results interacted with other Results and Options interacted with other Options. However, we have cases where we need interaction between the `option` type and `Result` or between `Result<T, Error_1>` type, and `Result<T, Error_2>`. In this recipe, you will learn how to build units to manage different error types and have them interact with each other; we will use our knowledge of combinators to achieve this.

Getting ready

We will require the Rust compiler and any text editor for developing the Rust code snippet.

How to do it...

Follow these steps to get through this recipe:

1. Create a file named `sample_multiple_err.rs` and open it in your text editor.
2. Write the code header with the relevant information:

```
|    //--- #####  
|    //--- Task: Handling multiple errors  
|    //--- Author: Vigneshwer.D  
|    //--- Version: 1.0.0  
|    //--- Date: 26 March 17  
|    #####
```

3. Define a generic alias named `Result<T>` for the `std::result::Result<T, String>` type:

```
|     type Result<T> = std::result::Result<T, String>;
```

4. Create a function named `double_first` that will accept the `vec` input and return a `Result<i32>` type:

```
| fn double_first(vec: Vec<&str>) -> Result<i32> {  
|     vec.first()  
|     .ok_or("Please use a vector with at least one  
|             element.".to_owned())  
|     .and_then(|s| s.parse::<i32>())  
|     .map_err(|e| e.to_string())  
|     .map(|i| 2 * i))  
| }
```

5. Create a function named `print` that will accept a `Result<i32>` type as input:

```
| fn print(result: Result<i32>) {  
|     match result {  
|         Ok(n) => println!("The first doubled is {}", n),  
|         Err(e) => println!("Error: {}", e),  
|     }  
| }
```

6. Define the `main` function and call the different functions:

```
| fn main() {  
|     let empty = vec![];  
|     let strings = vec![ "tofu", "93", "18" ];  
  
|     print(double_first(empty));
```

```
|     print(double_first(strings));  
| }
```

You will get the following output upon successful execution of the code:

```
viki@Vigneshwer:~/rust_cookbook/chapter6/code$ rustc -A warnings sample_multiple_err.rs  
viki@Vigneshwer:~/rust_cookbook/chapter6/code$ ./sample_multiple_err  
Error: Please use a vector with at least one element.  
Error: invalid digit found in string
```


How it works...

In the previous recipes, we handled cases where we had similar types, such as Results and Options interacting with other Results and Options. In this recipe, we handled cross types, for example, the interaction of Options with Results. We used our previous experience in relation to combinators to implement this.

First, we created an alias for `std::result::Result<T>, String>` using the `type` keyword as `Result<T>`, which we will use across the functional units of the code.

Let's check out the working of all the functional units:

- `double_first`: This function takes in the `vec` input and returns `Result<T>`. In our case, it took the first value of the vector sent to it using the `vec.first` method. If no values are provided, then it would enter `ok_or`, where it would print the statement asking the user to input at least one value to the vector. Next, it checks whether we are able to parse the string value to a valid integer value. If this is successful, it allows us to use the map function to double it by multiplying the parsed integer by 2; else, it takes the error value and maps it to the string equivalent.
- `print`: This function takes in `Result<T>`, and using the `match` statement, it checks whether we have an `ok` or `Err` case for printing the corresponding statement.

In the `main` function, we had vectors. Out of these, one was `empty`--there were no values in the vector--and the other was `string`, where we only had string values. When we call the `double_first` function with these input values, we get the corresponding errors.

Implementing early returns

Another way of dealing with different errors is using the combination of both `match` and `early return` statements. This is where we explicitly handle errors by returning them, and we do so without stopping the execution, as in the case of panic and so on.

Getting ready

We will require the Rust compiler and any text editor for developing the Rust code snippet.

How to do it...

Follow these steps to get through this recipe:

1. Create a file named `sample_early_ret.rs` and open it in your text editor.
2. Write the code header with the relevant information:

```
|    //--- #####  
|    //--- Task: Implementing early returns  
|    //--- Author: Vigneshwer.D  
|    //--- Version: 1.0.0  
|    //--- Date: 26 March 17  
|    #####
```

3. Define a generic alias named `Result<T>` for the `std::result::Result<T, String>` type:

```
|     type Result<T> = std::result::Result<T, String>;
```

4. Create a function named `double_first` that will accept the `vec` input and return a `Result<i32>` type:

```
| fn double_first(vec: Vec<&str>) -> Result<i32> {  
|     let first = match vec.first() {  
|         Some(first) => first,  
|         None => return Err("Please use a vector with at  
|             least  
|             one element.".to_owned())  
|     };  
  
|     match first.parse::<i32>() {  
|         Ok(i) => Ok(2 * i),  
|         Err(e) => Err(e.to_string()),  
|     }  
| }
```

5. Create a function named `print` that will accept a `Result<i32>` type as input:

```
| fn print(result: Result<i32>) {  
|     match result {  
|         Ok(n) => println!("The first doubled is {}", n),  
|         Err(e) => println!("Error: {}", e),  
|     }  
| }
```

6. Define the `main` function and call the different functions:

```
fn main() {
    let empty = vec![];
    let strings = vec!["tofu", "93", "18"];

    print(double_first(empty));
    print(double_first(strings));
}
```

You will get the following output upon successful execution of the code:

```
viki@Vigneshwer:~/rust_cookbook/chapter6/code$ rustc -A warnings sample_early_ret.rs
viki@Vigneshwer:~/rust_cookbook/chapter6/code$ ./sample_early_ret
Error: Please use a vector with at least one element.
Error: invalid digit found in string
```


How it works...

In the previous recipe, we explicitly handled errors using combinators. Rust also provides us with another way to deal with these cases, where we use a combination of `match` statements and early returns. The early returns is a way by which we can catch errors at an early stage of the function process and return back to the user of the application or library.

First, we created an alias for `std::result::Result<T, String>` using the `type` keyword as `Result<T>`, which we will use across the functional units of the code.

Let's check out the working of all the functional units:

- `double_first`: This function takes in the `vec` input and returns `Result<T>`. In our case, we used the early `return Err` to handle errors. We declared a variable named `first`, which held onto the first value of the vector that was passed. On this `first` variable, we performed `match` statements. If the value you get is `None`, use `return Err` to implement an early return to pass the error. In case you have string values in the vector element, use the `Err` value of the aliased type to raise the error.
- `print`: This function takes in `Result<T>`, and using the `match` statement, we check whether we have an `ok` or `Err` case for printing the corresponding statement.

In the `main` function, we had vectors in which one was `empty`, where there were no values in the vector. The other was `string`, where we had only string values when we called the `double_first` function. With this input, we get the corresponding errors.

Implementing the `try!` macro

We have reached a state where we can now avoid panicking, but explicitly handling all our errors is still a very difficult task. In this recipe, we'll use `try!` for cases where we simply need to unwrap without having to panic.

Getting ready

We will require the Rust compiler and any text editor for developing the Rust code snippet.

How to do it...

Follow these steps to get through this recipe:

1. Create a file named `sample_try.rs` and open it in your text editor.
2. Write the code header with the relevant information:

```
|    //--- #####  
|    //--- Task: Implementing try!  
|    //--- Author: Vigneshwer.D  
|    //--- Version: 1.0.0  
|    //--- Date: 26 March 17  
|    //--- #####
```

3. Define a generic alias `Result<T>` for the `std::result::Result<T, String>` type:

```
| type Result<T> = std::result::Result<T, String>;
```

4. Create a function named `double_first` that will accept the `Vec` input and return a `Result<i32>` type:

```
| fn double_first(vec: Vec<&str>) -> Result<i32> {  
|     let first = try!(vec.first())  
|     .ok_or("Please use a vector with at least one  
|             element.".to_owned());  
  
|     let value = try!(first.parse::<i32>()  
|                     .map_err(|e| e.to_string()));  
  
|     Ok(2 * value)  
| }
```

5. Create a function named `print` that will accept a `Result<i32>` type as input:

```
| fn print(result: Result<i32>) {  
|     match result {  
|         Ok(n) => println!("The first doubled is {}", n),  
|         Err(e) => println!("Error: {}", e),  
|     }  
| }
```

6. Define the `main` function and call the different functions:

```
| fn main() {  
|     let empty = vec![];  
|     let strings = vec!["tofu", "93", "18"];
```

```
    print(double_first(empty));
    print(double_first(strings));
}
```

You will get the following output upon successful execution of the code:

```
vik@Vigneshwer:~/rust_cookbook/chapter6/code$ rustc -A warnings sample_try.rs
vik@Vigneshwer:~/rust_cookbook/chapter6/code$ ./sample_try
Error: Please use a vector with at least one element.
Error: invalid digit found in string
```


How it works...

The `try!` macro enables us to simply unwrap without using panic. In the previous recipes, we used the unwrap and nested functionality many times to get the desired value. And `try!` is equivalent to an `unwrap` function that is returned instead of panic in the case of an error. In this recipe, you will learn how to use `try!` along with combinators.

First, we created an alias for `std::result::Result<T, String>` using the `type` keyword as `Result<T>`, which we will use across the functional units of the code.

Let's check out the working of all the functional units:

- `double_first`: This function takes in the `vec` type input and returns `aResult<T>` type. We declare a variable named `first` and assign it to `try!` macro statements for different cases. First, we check whether the first element of the vector is empty using `vec.first()`, which fetches the first value of the vector, if it's empty, we print a statement using the `ok_or` method and in the other `try!` we parse the `first` variable to an integer type. In case there is an error, we convert the error into a string using the `map_err` method.
- `print`: This function takes in `Result<T>`. Using the `match` statement, we check whether we have an `ok` or `Err` case for printing the corresponding statement.

In the main function, we had vectors in which one was `empty`, where there were no values in the vector. The other was `string`, where we only had string values. When we call the `double_first` function with these input values, we get the corresponding errors.

Defining your own error types

Rust allows us to define our own error types using custom Rust datatypes like `enum` and `struct`. We will create customized error-handling cases where we will able to define our own error types and have a definition for implementing or doing something to handle those error cases.

Getting ready

We will require the Rust compiler and any text editor for developing the Rust code snippet.

How to do it...

Follow these steps to get through this recipe:

1. Create a file named `sample_error.rs` and open it in your text editor.
2. Write the code header with the relevant information:

```
///-- #####  
///-- Task: Defining your own error type  
///-- Author: Vigneshwer.D  
///-- Version: 1.0.0  
///-- Date: 26 March 17  
///-- #####
```

3. Call the standard crates and create a generic alias type `Result<T>` for the `std::result::Result<T, CustomError>` type:

```
use std::num::ParseIntError;  
use std::fmt;  
  
type Result<T> = std::result::Result<T, CustomError>;
```

4. Create an `enum` type `CustomError`, which is our user-defined error type:

```
#[derive(Debug)]  
enum CustomError {  
    EmptyVec,  
    Parse(ParseIntError),  
}
```

5. Implement a customized way to display the error of the `CustomError` type:

```
impl fmt::Display for CustomError {  
    fn fmt(&self, f: &mut fmt::Formatter) -> fmt::Result {  
        match *self {  
            CustomError::EmptyVec =>  
                write!(f, "please use a vector with at least one  
                element"),  
            // This is a wrapper, so defer to the underlying  
            // types' implementation of `fmt`.  
            CustomError::Parse(ref e) => e(fmt(f),  
        }  
    }  
}
```

6. Create a function named `double_val` that will accept the `vec` input and return a `Result<i32>` type:

```
fn double_val(vec: Vec<&str>) -> Result<i32> {
    vec.first()
    // Change the error to our new type.
    .ok_or(CustomError::EmptyVec)
    .and_then(|s| s.parse::<i32>())
    // Update to the new error type here also.
    .map_err(CustomError::Parse)
    .map(|i| i * 2)
}
```

7. Create a function named `print` that will accept a `Result<i32>` type as input:

```
fn print(result: Result<i32>) {
    match result {
        Ok(n) => println!("The first doubled is {}", n),
        Err(e) => println!("Error: {}", e),
    }
}
```

8. Define the `main` function and call the different functions:

```
fn main() {
    let numbers = vec!["93", "18"];
    let empty = vec![];
    let strings = vec!["tofu", "93", "18"];

    print(double_val(numbers));
    print(double_val(empty));
    print(double_val(strings));
}
```

You will get the following output upon successful execution of the code:

```
viki@Vigneshwer:~/rust_cookbook/chapter6/code$ rustc -A warnings sample_error.rs
viki@Vigneshwer:~/rust_cookbook/chapter6/code$ ./sample_error
The first doubled is 186
Error: please use a vector with at least one element
Error: invalid digit found in string
```


How it works...

In general, we would define a good error type as something that would do the following things for the developer so that it's easy for them to understand where exactly the code is breaking at runtime:

- Represent the different errors in the code with the same type
- Display proper error functions to the user so it is easy for the developer to classify between different errors
- Hold proper information about the error

In this recipe, we created our own customized `enum` error type named `CustomError`. It had two types of data: `EmptyVec` and `Parse(ParseIntError)`. For each of these errors, we had customized implementation of error handling, where we produced customized error display messages for different errors so that our error type `CustomError` could follow all the preceding properties of a good error type. In the case of `EmptyVec`, we did not pass or need extra information about the error, but in the case of `Parse(ParseIntError)`, we had to supply the information required to parse the error implementation for its error.

Let's see how to implement custom display functions for different errors. In this case, we used the `impl` keyword to create a customized `fmt::Display` for our error type `CustomError`, where `fmt` represented the standard library. For the `fmt` method that takes in `&self, f: &mut fmt::Formatter` and returns the standard `fmt::Result`, we used the `match` statement to identify the type of error and display the corresponding error messages. In the case of `CustomError::EmptyVec`, we printed this error message: `please use a vector with at least one element`. In the case of `CustomError::Parse`, we formatted and printed the extra information of the type.

Let's check out the working of all the functional units:

- `double_first`: This function takes in the `vec` input and returns `Result<i32>`. It takes the first value of the vector sent to it using the `vec.first` method. If no values are provided, then it enters `ok_or`, where it changes

the error type to `CustomError::EmptyVec`. Next, it checks whether we are able to parse the string value to a valid integer value. If this is successful, we use the `map` function to double it by multiplying the parsed integer by `2`; else, it takes the error value and maps it to the other `CustomError::Parse` type.

- `print`: This function takes in `Result<i32>`. Using the `match` statement, we check whether we have an `ok` or `Err` case for printing the corresponding statement.

In the `main` function, we had vectors. One of them was a number, where we had the correct input type for producing the output without any errors. The next one was `empty`, where there were no values in the vector. Then, there was `strings`, where we had the first value as a string of character values that couldn't be parsed to an integer. When we call the `double_first` function with these input values, we get the corresponding errors.

Implementing the boxing of errors

Rust allows us to box our error types, which is the process of creating wrapper error types around standard library error types. Boxing of errors is a common practice where developers bring together all the error types of the different libraries and use them to build the project.

Getting ready

We will require the Rust compiler and any text editor for developing the Rust code snippet.

How to do it...

Follow these steps to get through this recipe:

1. Create a file named `sample_box.rs` and open it in your text editor.
2. Write the code header with the relevant information:

```
//-- #####  
//-- Task: Implementing Boxing  
//-- Author: Vigneshwer.D  
//-- Version: 1.0.0  
//-- Date: 26 March 17  
//-- #####
```

3. Call the standard crates and create a generic alias type, namely `Result<T>`, for the `std::result::Result<T, Box<error::Error>>` type:

```
use std::error;  
use std::fmt;  
use std::num::ParseIntError;  
  
type Result<T> = std::result::Result<T, Box<error::Error>>;
```

4. Create an `enum` type `CustomError`, which would be our user-defined error type:

```
#[derive(Debug)]  
enum CustomError {  
    EmptyVec,  
    Parse(ParseIntError),  
}
```

5. Convert the standard library error type into a custom type:

```
impl From<ParseIntError> for CustomError {  
    fn from(err: ParseIntError) -> CustomError {  
        CustomError::Parse(err)  
    }  
}
```

6. Implement a customized way to display an error for the `CustomError` type:

```
impl fmt::Display for CustomError {  
    fn fmt(&self, f: &mut fmt::Formatter) -> fmt::Result {  
        match *self {
```

```

CustomError::EmptyVec =>
write!(f, "please use a vector with at least one element"),
CustomError::Parse(ref e) => e fmt(f),
}
}
}
}

```

7. Implement the `error` trait for the `CustomError` type:

```

impl error::Error for CustomError {
fn description(&self) -> &str {
match *self {
CustomError::EmptyVec => "empty vectors not allowed",
CustomError::Parse(ref e) => e.description(),
}
}
fn cause(&self) -> Option<&error::Error> {
match *self {
CustomError::EmptyVec => None,
CustomError::Parse(ref e) => Some(e),
}
}
}

```

8. Create a function named `double_first` that will accept the `vec` input and return a `Result<i32>` type:

```

fn double_val(vec: Vec<&str>) -> Result<i32> {
let first = try!(vec.first().ok_or(CustomError::EmptyVec));
let parsed = try!(first.parse::<i32>());
Ok(2 * parsed)
}

```

9. Create a function named `print` that will accept a `Result<i32>` type as input:

```

fn print(result: Result<i32>) {
match result {
Ok(n) => println!("The first doubled is {}", n),
Err(e) => println!("Error: {}", e),
}
}

```

10. Define the `main` function and call the different functions:

```

fn main() {
let numbers = vec!["93", "18"];
let empty = vec![];
let strings = vec!["tofu", "93", "18"];

print(double_val(numbers));
print(double_val(empty));
}

```

```
|     print(double_val(strings));  
| }
```

You will get the following output upon successful execution of the code:

```
viki@Vigneshwer:~/rust_cookbook/chapter6/code$ rustc -A warnings sample_box.rs  
viki@Vigneshwer:~/rust_cookbook/chapter6/code$ ./sample_box  
The first doubled is 186  
Error: please use a vector with at least one element  
Error: invalid digit found in string
```


How it works...

The `std` library automatically converts any type that implements the `Error` trait into the trait object `Box<Error>` via the `From` trait object. However, a user may use many external libraries, and different libraries provide their own error types. In order to define a valid `Result<T, E>` type, perform the following tasks:

- Define a new wrapper error type around the library's standard error types
- Convert the error types into `String` or any other type that is convenient to handle
- Box the error types into the `Box<Error>` type

In this recipe, we started off by calling the standard libraries `std::error`, `std::fmt`, and `std::num::ParseIntError`. We then created an alias `Result<T>` for `std::result::Result<T, Box<error::Error>>`. Next, we created our own customized `enum` error type named `CustomError`, which had two types of data: `EmptyVec` and `Parse(ParseIntError)`.



The compiler is capable of providing basic implementations for some traits via the `#[derive]` attribute, where an attribute is a metadata applied to some module, crate, or item. We use the `#[derive(Debug)]` for getting an output format that is programmer-facing and has more debugging context.

We converted the standard library's `ParseIntError` error into the custom error type `CustomError` by implementing the `From` trait. We did this because the `from` method takes in the standard error type `ParseIntError` as `err` and returns the `CustomError` type by setting `CustomError::Parse(err)`.

Now let's see how we implemented custom display functions for the different errors. We used the `impl` keyword to create a customized `fmt::Display` for our `CustomError` error type, where `fmt` represented the standard library. The `fmt` method takes in `&self, f: &mut fmt::Formatter` and returns the

standard `fmt::Result`. We used the `match` statement to identify what type of error it was and display the corresponding error messages. In the case of `CustomError::EmptyVec`, we printed this error message: `please use a vector with at least one element`. In the case of `CustomError::Parse`, we formatted and printed the extra information of the type.

To implement `Box<Error>`, we had to implement the `Error` trait where we had two methods: `description` and `cause`. These methods take the value of the trait and return them. In the `description` method, using the `match` statement, we assigned a description about the error types; here we matched

`CustomError::EmptyVec` to empty vectors not allowed and `CustomError::Parse(ref e)` to `e.description()`. Similarly, in the case of `cause`, we had sample values that led to the error, and we matched `CustomError::EmptyVec` to `None` and

`CustomError::Parse(ref e)` to `Some(e)`.

Let's check out the working of all the functional units:

- `double_first`: This function takes in the `vec` input and returns `Result<i32>`. In our case, it took the first value of the vector sent to it using the `vec.first` method with the `try!` macro and assigned it to the `first` variable. If no values are provided, then it would enter `ok_or`, where it changes the error type to `CustomError::EmptyVec`. Next, we checked whether we were able to parse the string value to a valid integer value by `first.parse::<i32>()`, using the `try!` macro and assigning it to the `parsed` variable. If this is successful, we double it by multiplying the parsed integer by `2`. In the `ok` data type, it returns the `enum` type `Result<i32>`; else, it takes the error value parsed by the error type.
- `print`: This function takes in `Result<i32>`. Using the `match` statement, we check whether we have an `ok` or `Err` case to print the corresponding statement.

In the `main` function, we had vectors. Among these, one was a number, where we had the correct input type for producing the output without any errors. The next one was `empty`, meaning there were no values in the vector. The other one was `strings`, where we had the first value of a string of character

values that couldn't be parsed into an integer. When we call the `double_first` function with these input values, we get the corresponding errors.

Hacking Macros

In this chapter, we will be covering the following recipes:

- Building macros in Rust
- Implementing matching in macros
- Playing with common Rust macros
- Implementing designators
- Overloading macros
- Implementing repeat
- Implementing DRY

Introduction

Until now, we have seen many statements in Rust ending with an exclamation mark (!), such as `println!`, `try!`, and so on. These commands have performed powerful operations to execute specific tasks. Rust provides a powerful macro system that allows metaprogramming. Macros look like functions but their names end with an exclamation mark (!). Macros are expanded into source code which, gets compiled into the program. In this recipe, we will look into the various aspects of macros, ranging from defining your own application-specific macros to testing them.

Building macros in Rust

In this recipe, we will learn about `macro_rules!`--the syntax that will help us define our custom application-specific macro, which can have a unique name according to the application terminology.

Getting ready

We will require the Rust compiler and any text editor to develop the Rust code snippet.

How to do it...

Follow the given steps to implement this recipe:

1. Create a file named `sample_macro.rs`, and open it in your text editor.
2. Write the code header with the relevant information:

```
//-- #####  
//-- Task: Building your first macro in Rust  
//-- Author: Vigneshwer.D  
//-- Version: 1.0.0  
//-- Date: 26 March 17  
//-- #####
```

3. Create a macro named `Welcome_RustBook`:

```
// This is a simple macro named `say_hello`.  
macro_rules! Welcome_RustBook {  
    () => (  
        // The macro will expand into the contents of this block.  
        println!("Welcome to Rust Cookbook!");  
    )  
}
```

4. Define the `main` function and call the `Welcome_RustBook` macro:

```
fn main() {  
    // This call will expand into `println!("Hello");`  
    Welcome_RustBook!()  
}
```

We will get the following output on the successful execution of our code:

```
viki@Vigneshwer:~/rust_cookbook/chapter7/code$ rustc -A warnings sample_macro.rs  
viki@Vigneshwer:~/rust_cookbook/chapter7/code$ ./sample_macro  
Welcome to Rust Cookbook!
```


How it works...

We use the `macro_rules!` macro to create a custom-named macro; here, we made a macro named `Welcome_RustBook!`. The general syntax of `macro_rules!` is as follows:

```
| macro_rules! macro_name { ... }
```

Inside the `macro_rules!` macro, we match the arguments. In the case of this recipe, we do not accept any arguments from the user, so we match `() => (` (a certain set of action items). The empty parentheses, `()`, in the code indicate that the macro takes no argument. The macro will expand into the contents of the block of no arguments at compile time, where we have `println!("Welcome to Rust Cookbook!");`, which basically prints a default statement.

In the main function, we call `Welcome_RustBook!` macro in the function definition, just like we would call any other macro. We will see the default statement printed in the terminal.

Implementing matching in macros

Let's go ahead and make our macro a bit more complex by adding more rules in our macro, the rules are basically pattern matching cases. In this recipe, the key takes-away will be to learn how we can define pattern matching cases in macro rules.

Getting ready

We will require the Rust compiler and any text editor to develop the Rust code snippet.

How to do it...

Follow the mentioned steps to implement this recipe:

1. Create a file named `sample_match.rs`, and open it in your text editor.
2. Write the code header with the relevant information:

```
//-- #####  
//-- Task: Implement matching  
//-- Author: Vigneshwer.D  
//-- Version: 1.0.0  
//-- Date: 26 March 17  
//-- #####
```

3. Create a macro named `Check_Val`:

```
macro_rules! Check_Val {  
    ($x => $e:expr) => (println!("mode X: {}", $e));  
    ($y => $e:expr) => (println!("mode Y: {}", $e));  
}
```

4. Define the `main` function and call the `Check_Val` macro:

```
fn main() {  
    Check_Val!(y => 3);  
}
```

We will get the following output on the successful execution of our code:

```
[viki@Vigneshwer:~/rust_cookbook/chapter7/code$ rustc -A warnings sample_match.rs  
viki@Vigneshwer:~/rust_cookbook/chapter7/code$ ./sample_match  
mode Y: 3
```


How it works...

In this recipe, we create a macro named `Check_Val!`, which basically plays the role as a `match` expression arm, but the matching happens through the Rust syntax trees during compilation time. The common syntax of a pattern is as follows:

```
| ( $( $x:expr ),* ) => { ... };
```

Here, the term *pattern* refers to the left-hand side of `=>`, which is known as *matcher* in Rust.

The `$x:expr` matcher will match any Rust expression and will bind that to syntax tree to the metavariable `$x`, any Rust tokens that appear in a matcher must match exactly.

We have two pattern matching cases here: `x => $e:expr` and `y => $e:expr`. The metavariable is `$e`, which is used in macro definitions for operations to be done following the successful pattern matching of a macro rule. When we call `Check_Val!(y => 3);` in the `main` function, the output is `mode Y:3`. Here the second case is passed and the value of `$e` is the same as that of the arguments passed to the `Check_Val!` macro in the `main` function.



If we had called `Check_Val!(z => 3);` we would have got `error: no rules expected the token 'z'`, as we haven't defined a rule for the token `z` and surrounding the matcher with `$(...),` will match zero or more expressions, separated by commas.*

Playing with common Rust macros

Throughout this book, we have defined and used common Rust macros that will help us perform basic operations, such as printing, and so on. Rust offers these macros by default, as these are very complex to be implemented by the user. In this recipe, we will learn a few common Rust macros.

Getting ready

We will require the Rust compiler and any text editor to develop the Rust code snippet.

How to do it...

Follow the given steps to implement this recipe:

1. Create a file named `sample_common_macros.rs`, and open it in your text editor.
2. Write the code header with the relevant information:

```
//-- #####  
//-- Task: Implementing common macros in rust  
//-- Author: Vigneshwer.D  
//-- Version: 1.0.0  
//-- Date: 26 March 17  
//-- #####
```

3. Create the `main` function where we implement a few inbuilt standard Rust macros:

```
fn main() {  
  
    // Creating a vector  
    let v = vec![1, 2, 3, 4, 5];  
    print!("Vector :- {:?}", v);  
  
    // Macros used for testing  
    assert!(true);  
    assert_eq!(5, 3 + 2);  
  
    // assert!(5 < 3);  
    // assert_eq!(5, 3);  
  
    // Gives a message to panic  
    // panic!("oh no!");  
}
```

We will get the following output on successful execution of our code:

```
[vigneshwers-MacBook-Air:code vigneshwerdhinakaran$ rustc -A warnings sample_common_macros.rs  
[vigneshwers-MacBook-Air:code vigneshwerdhinakaran$ ./sample_common_macros  
Vector :- [1, 2, 3, 4, 5][vigneshwers-MacBook-Air:code vigneshwerdhinakaran$
```


How it works...

We declared all the standard macros in the main function. Let's deep dive into each one of them in the following order:

- We used the `vec!` macro to create a vector in Rust. It creates `Vec<T>`.
- The next two macros are extensively used in tests: the first one is `assert!`, which takes a Boolean value to pass and the second one is `assert_eq!`, which takes two values and checks for their equality. The true value passes and the false one leads to the `panic!` macro, which causes the thread to panic or break.

In this recipe, we have used the `vec!` macro to create a vector, `v`. The conditions inside the `assert!` and `assert_eq!` macros pass. The failure cases have been commented out, as they would cause panic during runtime.

Implementing designators

Rust provides a list of designators, which help us create units, such as functions, and execute expressions in macros.

Getting ready

We will require the Rust compiler and any text editor to develop the Rust code snippet.

How to do it...

Follow the mentioned steps to implement this recipe:

1. Create a file named `sample_designator.rs`, and open it in your text editor.
2. Write the code header with the relevant information:

```
//-- #####  
//-- Task: Implementing designator  
//-- Author: Vigneshwer.D  
//-- Version: 1.0.0  
//-- Date: 26 March 17  
//-- #####
```

3. Create a macro named `create_function`, which accepts a designator as an argument:

```
macro_rules! create_function {  
    ($func_name:ident) => (  
        fn $func_name() {  
            // The `stringify!` macro converts an `ident`  
            // into a string.  
            println!("You called {:?}()",  
                stringify!($func_name))  
        }  
    )  
}
```

4. Call the `create_function` macro to create two functions, `foo` and `bar`:

```
create_function!(foo);  
create_function!(bar);
```

5. Create a macro named `print_result`:

```
macro_rules! print_result {  
    ($expression:expr) => (  
        println!("{:?} = {:?}",  
            stringify!($expression),  
            $expression)  
    )  
}
```

6. Define the `main` function, where we play around with the macros we created:

```
fn main() {
    foo();
    bar();

    print_result!(1u32 + 1);

    // Recall that blocks are expressions too!
    print_result!({
        let x = 1u32;

        x * x + 2 * x - 1
    });
}
```

We will get the following output on successful execution of our code:

```
viki@Vigneshwer:~/rust_cookbook/chapter7/code$ rustc -A warnings sample_designator.rs
viki@Vigneshwer:~/rust_cookbook/chapter7/code$ ./sample_designator
You called "foo"()
You called "bar"()
"1u32 + 1" = 2
"[ let x = 1u32; x * x + 2 * x - 1 ]" = 2
```


How it works...

In general, the arguments of a macro are prefixed by a dollar sign (\$) and a type annotated with a designator. Here, in this recipe, we have used two commonly used designators, which are `expr`, used for expressions, and `ident`, which is used for variable/function names.

Let's understand the two main macros that we have created to implement the designator in the Rust code:

- `create_function!`: This macro takes an argument of the `ident` designator and creates a function named `$func_name`, which is used across the code for creating the function. The `ident` designator, as mentioned previously, is used for variable/function names. Inside the block of the `($func_name:ident)`, pattern, we define the function `fn $func_name`, and we have the `stringify!` macro in its body, which converts a `$func_name` into a string.
- `print_result!`: This macro takes an expression of the `expr` type and prints it as a string along with its result. The `expr` designator is used for expressions. In the block for the expression pattern, we use the `stringify!` macro, which converts the expression into a string and also executes it.

We create functions named `foo` and `bar` with the preceding macro by using `create_function!(foo);` and `create_function!(bar);`. In the `main` function, where we called the two functions, that is, `foo` and `bar`, which return the string. We call `function_name`. Next, we call `print_result!`, with a block of expression as an argument, where we create a variable, `x`, and assign it a value of `1u32`, which is a 32-bit unsigned integer type. We then run `x * x + 2 * x - 1`, which gives us the output of `2`.

Overloading macros

Overloading macros in Rust is the process of providing multiple combinations of similar arguments, where we expect the macro to handle them and provide custom results according to the combination passed.

Getting ready

We will require the Rust compiler and any text editor to develop the Rust code snippet.

How to do it...

Follow the given steps to implement this recipe:

1. Create a file named `sample_overloading_macros.rs`, and open it in your text editor.
2. Write the code header with the relevant information:

```
//-- #####  
//-- Task: Implementing  
//-- Author: Vigneshwer.D  
//-- Version: 1.0.0  
//-- Date: 26 March 17  
//-- #####
```

3. Create a macro named `test`, for which we will implement overloading:

```
macro_rules! test {  
  
    ($left:expr; and $right:expr) => (  
        println!("{} and {} is {}", stringify!($left),  
               stringify!($right),  
               $left && $right)  
    );  
  
    ($left:expr; or $right:expr) => (  
        println!("{} or {} is {}", stringify!($left),  
               stringify!($right),  
               $left || $right)  
    );  
}
```

4. Define the `main` function in which we'll implement the features of the macro:

```
fn main() {  
    test!(1i32 + 1 == 2i32; and 2i32 * 2 == 4i32);  
    test!(true; or false);  
}
```

We will get the following output on the successful execution of our code:

```
viki@Vigneshwer:~/rust_cookbook/chapter7/code$ rustc -A warnings sample_overloading_macros  
viki@Vigneshwer:~/rust_cookbook/chapter7/code$ ./sample_overloading_macro  
"1i32 + 1 == 2i32" and "2i32 * 2 == 4i32" is true  
"true" or "false" is true
```


How it works...

In this recipe, we create a macro named `test`, which takes two `expr` designators used for taking an expression as arguments and assigning it to two variables, `$left` and `$right`, where `$left` is assigned to the first expression and `$right`, to the second expression.

Inside the macros, we have two rules, which are as follows:

- `($left:expr; and $right:expr)`: In this rule, we want to return a Boolean value. Here, we evaluate both the expressions and pass the values to the `&&` operator.
- `($left:expr; or $right:expr)`: In this rule, we want to return a Boolean value. Here, we evaluate both the expressions and pass the values to the `||` operator.



Arguments don't need to be separated by a comma and each arm must end with a semicolon.

In the `main` function, we call the `test!` macro two times with different arguments, where we have the combinations. The `test!(1i32 + 1 == 2i32; and 2i32 * 2 == 4i32);` combination returns the string form of the expression along with the result, which is `true`; `test!(true; or false);` similarly returns `true`.

Implementing repeat

Repeat is the ability of a particular macro to accept arguments that repeat at least once. In this recipe, you will learn the syntax to implement repeat in Rust.

Getting ready

We will require the Rust compiler and any text editor to develop the Rust code snippet.

How to do it...

Follow the given steps to implement this recipe:

1. Create a file named `sample_repeat.rs`, and open it in your text editor.
2. Write the code header with the relevant information:

```
//-- #####  
//-- Task: Implementing repeat  
//-- Author: Vigneshwer.D  
//-- Version: 1.0.0  
//-- Date: 26 March 17  
//-- #####
```

3. Create a macro named `find_min`, in which we implement `repeat`:

```
macro_rules! find_min {  
    // Base case:  
    ($x:expr) => ($x);  
    // '$x` followed by at least one '$y,  
    // Call `find_min!` on the tail '$y'  
    ($x:expr, $($y:expr),+) => (  
        std::cmp::min($x, find_min!($($y),+))  
    )  
}
```

4. Create a `main` function in which we pass multiple arguments to `find_min`:

```
fn main() {  
    println!("{}", find_min!(1u32));  
    println!("{}", find_min!(1u32 + 2, 2u32));  
    println!("{}", find_min!(5u32, 2u32 * 3, 4u32));  
}
```

We will get the following output on the successful execution of our code:

```
viki@Vigneshwer:~/rust_cookbook/chapter7/code$ rustc -A warnings sample_repeat.rs  
viki@Vigneshwer:~/rust_cookbook/chapter7/code$ ./sample_repeat  
1  
2  
4
```


How it works...

Macros can use `+` in the argument list to indicate that an argument may repeat at least once or `*` to indicate that the argument may repeat zero or more times.

In the recipe, we have a macro named `find_min`, which has two rules in which the matcher with `$(...), +` will match one or more expressions, separated by commas. In the first case, we have `($x:expr)`, which just executes the expression and returns the output; this will be matched if we pass only one expression to the `find_min` macro. In the second case, we have `($x:expr, $($y:expr), +)`. Here, `$x` is followed by at least one `$y`, and inside the block, we call the `find_min!` macro on the tail `$y`; these values are fed to `std::cmp::min`, which returns the smallest value from the argument list. On the second call, it would execute the first case of the macro and return the expression.

In the `main` function, we run the following cases and print the results:

- `find_min!(1u32)`: This will execute the first case and return `1`
- `find_min!(1u32 + 2, 2u32)`: This will go to the second case, where the macro will be called again for the second expression and the `min` result of those two expressions will be returned, which is `2`
- `find_min!(5u32, 2u32 * 3, 4u32)`: This is similar to the second case, but here the macro will be called two times and the `min` result of all the expressions will be returned, which is `4` in this case

Implementing DRY

Using Don't Repeat Yourself (DRY), in this recipe, we are going to create a test case for some basic standard arithmetic operations in Rust. The catch is, however, that we are going to work on automating them using macros and their features so that we can reduce redundant code.

Getting ready

We will require the Rust compiler and any text editor to develop the Rust code snippet.

How to do it...

Follow the given steps to implement this recipe:

1. Create a file named `sample_dry.rs`, and open it in your text editor.
2. Write the code header with the relevant information:

```
|    //--- #####  
|    //--- Task: Implementing  
|    //--- Author: Vigneshwer.D  
|    //--- Version: 1.0.0  
|    //--- Date: 26 March 17  
|    //--- #####
```

3. Call the standard operation crate:

```
| use std::ops::{Add, Mul, Sub};
```

4. Create a macro named `assert_equal_len`:

```
| macro_rules! assert_equal_len {  
|     ($a:ident, $b: ident, $func:ident, $op:tt) => (  
|         assert!($a.len() == $b.len(),  
|             "{:?}: dimension mismatch: {:?} {:?} {:?}",  
|             stringify!($func),  
|             ($a.len(),),  
|             stringify!($op),  
|             ($b.len(),));  
|     )  
| }
```

5. Create a macro named `op`:

```
| macro_rules! op {  
|     ($func:ident, $bound:ident, $op:tt, $method:ident) => (  
|         fn $func<T: $bound<T, Output=T> + Copy>(&mut xs: &mut Vec<T>, ys:  
|             &Vec<T>) {  
|             assert_equal_len!(xs, ys, $func, $op);  
|             for (x, y) in xs.iter_mut().zip(ys.iter()) {  
|                 *x = $bound::$method(*x, *y);  
|                 // *x = x.$method(*y);  
|             }  
|         }  
|     )  
| }
```

6. Implement the `add_assign`, `mul_assign`, and `sub_assign` functions:

```
op! (add_assign, Add, +=, add);
op! (mul_assign, Mul, *=, mul);
op! (sub_assign, Sub, -=, sub);
```

7. Create a module named test:mod test {:

```
use std::iter;
macro_rules! test {
    ($func: ident, $x:expr, $y:expr, $z:expr) => {
        #[test]
        fn $func() {
            for size in 0usize..10 {
                let mut x: Vec<_> =
                    iter::repeat($x).take(size).collect();
                let y: Vec<_> = iter::repeat($y).take(size).collect();
                let z: Vec<_> = iter::repeat($z).take(size).collect();

                super::$func(&mut x, &y);

                assert_eq!(x, z);
            }
        }
    }

    // Test `add_assign`, `mul_assign` and `sub_assign`
    test!(add_assign, 1u32, 2u32, 3u32);
    test!(mul_assign, 2u32, 3u32, 6u32);
    test!(sub_assign, 3u32, 2u32, 1u32);
}
```

We will get the following output on the successful execution of our code:

```
vik@Vigneshwer:~/rust_cookbook/chapter7/code$ rustc --test sample_dry.rs
vik@Vigneshwer:~/rust_cookbook/chapter7/code$ ./sample_dry

running 3 tests
test test::sub_assign ... ok
test test::mul_assign ... ok
test test::add_assign ... ok

test result: ok. 3 passed; 0 failed; 0 ignored; 0 measured
```


How it works...

Macros allow developers to write DRY code by factoring out the common parts of functions and/or test suites. In this recipe, we implemented tested for the `+=`, `*=`, and `-=` operators on `Vec<T>`. We have used a new designator in this recipe, `tt`; which stands for token tree and is used for operators and tokens.

Let's first understand all the functional macro units in the code:

- `assert_equal_len`: This macro takes in four arguments as inputs, which are `$a`, `$b`, and `$func`, of the `ident` type, and `$op` of the `tt` type. If the macro receives these arguments, then it will check whether `$a` and `$b` are of the same length by using the `len()` method inside the `assert!` macro, which will return a Boolean value of `true` in the case of success or else, it prints a failure statement saying `dimension mismatch`.
- `op`: This macro takes in four arguments as input, which are `$func`, `$bound`, and `$method`, of the `ident` type, and `$op` of the `tt` type. We create the corresponding operator function with this macro, where `$func` is the name of the function and is the first argument in the list, with two arguments of the `Vec<T>` type: `xs` and `ys`. Both the variables are shared with the macros, and `xs` is provided with a mutable permission while it is shared. Inside the function, we perform the operation with `$bound::$method` for all the values of the vectors `xs` and `ys`, and the results are stored in `x` as it has mutable access. Here, `$bound` is the standard module and its `$method` corresponds to its unit. With this macro, we are able to perform a lot of methods on the data passed, which reduces the code.
- `test`: This macro takes in four arguments as input, which are `$func`, of the `ident` type, and `$x`, `$y`, and `$z`, which are `expr` of the `ident` type, and are present inside the `test` module, which is invoked while we run our test cases. Inside the `test` macro, we create the function with the name of `$func`. By doing so, it will become a function or unit of the parent `test` module. We iterate across the values to create vectors in which we perform `super:::$func(&mut x, &y)`. The `super` here refers to the function

that we created by using the `op` macro, which updates the value of `x` based on the operation we wanted to perform. In the last step, we validate `test` by comparing the updated `x` vector with the `z` vector, which is the desired value. The `assert_eq!` macro will return `true` if the values match; else it will panic out.

In this code, we use a certain set of standard libraries, which are `ops` and `item`. First, we create the different operations that we want to implement, so we call the `op!` and create `add_assign`, `mul_assign`, and `sub_assign`. Later in the test module, we call the test case for the different functions that we have created. Here, we give all the cases for passing and run the `--test` option during compilation to run the test cases.

Integrating Rust with Other Languages

We will be covering the following recipes in this chapter:

- Calling C operations from Rust
- Calling Rust commands from C
- Calling Rust operations from Node.js apps
- Calling Rust operations from Python
- Writing a Python module in Rust

Introduction

In this chapter, we will cover the techniques and steps that will help us create Rust units in our existing applications that are written in other languages, such as C, Node.js, and Python. We will deep dive into the concept of foreign function interface, which helps us in writing bindings for a foreign code base. Rust outperforms many programming languages in the aspect of stability and safer parallelism. It would be ideal for production level application developers to try out Rust for building small units in the code, and test to see if there is a considerable amount of performance change. This chapter helps to facilitate these kinds of thoughts and ideas.

Calling C operations from Rust

In this recipe, we will call an external C function from a Rust application. This technique can be really helpful when a developer wants to use some project-specific C library dependencies in the Rust code.

Getting ready

We will have to install the following crate before we can go ahead and create the binding between Rust and C. Follow the given steps to download and set up the `libc` crate, `gcc` crate, and `gcc` compiler for your project:

1. We will require the Rust compiler and any text editor to develop the Rust code snippet.
2. Create a Rust library project using the Cargo tool:

```
| cargo new --bin rust-to-c
```

3. Enter the newly created Rust library project:

```
| cd rust-to-c
```

4. Open the `cargo.toml` file in your favorite text editor and make the modification shown in the following screenshot. Here, we'll use `libc` version `0.1` and `gcc` version `0.3`, and we'll also mention the `build` script, which is `build.rs`:

```
[package]
name = "rust-to-c"
version = "0.1.0"
authors = ["Vigneshwer.D <dvigneshwer@gmail.com>"]
build = "build.rs"

[dependencies]
libc = "0.1"

[build-dependencies]
gcc = "0.3"
```

5. Install the `gcc` compiler on your machine, which is usually present by default:

```
| sudo apt-get install gcc
```


How to do it...

Follow the given steps to implement this recipe:

1. Create a file named `build.rs` in the root location of the project and open it your text editor.
2. Write the code header with the relevant information:

```
//-- #####  
//-- Task: Build script  
//-- Author: Vigneshwer.D  
//-- Version: 1.0.0  
//-- Date: 14 April 17  
//-- #####
```

3. Enter the following code in the script:

```
extern crate gcc;  
  
fn main() {  
    gcc::Config::new().file("src/double.c")  
        .compile("libdouble.a");  
}
```

4. Change to the `src` directory in the project:

```
| cd src
```

5. Create the `double.c` script, which is the C application:

```
| touch double.c
```

6. Create a `double_input` function in the `double.c` application, which we will use in the Rust application:

```
int double_input(int input) {  
    return input * 2;  
}
```

7. Enter the `main.rs` script and enter the following code:

```
extern crate libc;  
  
extern {  
    fn double_input(input: libc::c_int) -> libc::c_int;  
}
```

```
fn main() {
    let input = 4;
    let output = unsafe { double_input(input) };
    println!("{} * 2 = {}", input, output);
}
```

We will get the following output on the successful execution of our code:

```
viki@Vigneshwer:~/rust_cookbook/chapter8/code/rust-to-c$ cargo run
  Compiling libc v0.1.12
  Compiling gcc v0.3.45
  Compiling rust-to-c v0.1.0 (file:///home/viki/rust_cookbook/chapter8/code/rust-to-c)
    Finished debug [unoptimized + debuginfo] target(s) in 2.36 secs
      Running `target/debug/rust-to-c`
4 * 2 = 8
```


How it works...

In this recipe, we created a Rust project that can be used as a third-party package C application in the Rust code.

We used the `libc` crate, which is used as a library for types and bindings to native C functions that are often found in other common platform libraries. This project dependency is mentioned in the `Cargo.toml` file under the `dependencies` field, and, then, in the `build-dependencies` section of the manifest, we have `gcc = 0.3`, which is the dependency of the `build` script.



The `build` script does not have the access to dependencies listed in the `dependencies` of the `Cargo.toml` manifest section. The `build` dependencies will not be available to the package files; this is done by the Cargo tool so that the package and the build script are compiled separately, so their dependencies need not coincide.

Rust provides a `build` script support where some packages need to compile third-party non-Rust code; for example, in our case we have a C script called `double.c`. The packages needed to link to C script, which can either be located on the system or possibly need to be built from source. Cargo does not replace other tools for building these packages, but it integrates them with the `build` configuration option, like, in our case, in the `package` section of the manifest, where we have a file named `build` with the `build.rs` script.

The Rust file designated by the `build` command, which, in our case, is `build.rs`, will be compiled first, before anything else is compiled in the package. This allows your Rust code to depend on the built or generated artifacts. In the `build.rs` script, we are building the native C code as part of the package; this package will later be used by the Rust code. We use an external crate named `gcc`, which invokes the externally maintained C compiler.

The `build` script starts out in the `main` function where

```
gcc::Config::new().file("src/double.c").compile("libdouble.a")
```

starts off by compiling our C file into an object file (by invoking `gcc`) and then converting the object file (`double.o`) into a static library (`libdouble.a`). The object file is stored in `target/debug/build/<package>/out` location.

In the `src` directory, we have the Rust project package file in which we have the native C script, `double.c`. The `double.c` script has a function named `double_input`, which takes in an integer argument named `input` and returns `input * 2`, which basically doubles the value passed.

In `main.rs`, we first import the `libc` crate and then define the function signature in the `extern` block (since it's a third-party package) as `fn double_input(input: libc::c_int) -> libc::c_int`. Here, we use the `libc` type so that there is a smooth conversion of types between Rust and C, which we do not have to handle while calling the foreign function, `double_input`. From the `main` function, we place it in the `unsafe` block.



The `extern` block is a list of function signatures in a foreign library and the foreign functions are assumed to be unsafe. So, when we call them, they need to be wrapped with the `unsafe` block as a promise to the compiler that everything contained within is truly safe.

Calling Rust commands from C

In this recipe, we will perform the exact opposite operation of the last recipe. Here, we will call a Rust function from the C script. This technique can be really helpful when the developer wants to build a specific Rust unit in their C project.

Getting ready

We will have to install the following system dependencies before we can go ahead and create a Rust unit for the C project. Follow the given steps to download and install the `build-essential` package for your project:

1. We will require the Rust compiler and any text editor to develop the Rust code snippet.
2. Create a Rust library project using the Cargo tool:

```
| cargo new c-to-rust
```

3. Enter the newly created Rust library project:

```
| cd c_to_rust
```

4. Open the `cargo.toml` file in your favorite text editor and make the modification shown in the following screenshot:

```
[package]
name = "c-to-rust"
version = "0.1.0"
authors = ["Vigneshwer.D <dvigneshwer@gmail.com>"]

[lib]
name = "double_input"
crate-type = ["staticlib"]
```

5. Install the `build-essential` tool on your machine, which is usually present by default:

```
| sudo apt-get update && apt-get install build-essential
```


How to do it...

Follow the given steps to implement this recipe:

1. Create a file named `lib.rs` and open it in your text editor.
2. Write the code header with the relevant information:

```
///-- #####  
///-- Task: Rust Function for  
///-- Author: Vigneshwer.D  
///-- Version: 1.0.0  
///-- Date: 14 April 17  
///-- #####
```

3. Create the `double_input` function in `lib.rs` Rust script with the given attributes:

```
#[crate_type = "staticlib"]  
  
#[no_mangle]  
pub extern fn double_input(input: i32) -> i32 {  
    input * 2  
}
```

4. Create the file `main.c` in the `c_to_rust/src` directory, which is the C script:

```
#include <stdint.h>  
#include <stdio.h>  
  
extern int32_t double_input(int32_t input);  
  
int main() {  
    int input = 4;  
    int output = double_input(input);  
    printf("%d * 2 = %d\n", input, output);  
    return 0;  
}
```

5. Create `Makefile` in the root location of the project directory, `c_to_rust`, for creating the `build` rules:

```
ifeq ($(shell uname), Darwin)  
LDFLAGS := -Wl,-dead_strip  
else  
LDFLAGS := -Wl,--gc-sections -lpthread  
endif  
  
all: target/double
```

```
target/double

target:
mkdir -p $@

target/double: target/main.o
target/debug/libdouble_input.a
$(CC) -o $@ $^ $(LDFLAGS)

target/debug/libdouble_input.a: src/lib.rs Cargo.toml
cargo build

target/main.o: src/main.c | target
$(CC) -o $@ -c $<

clean:
rm -rf target
```

We will get the following output on the successful execution of our code:

```
viki@Vigneshwer:~/rust_cookbook/chapter8/code/c-to-rust$ make -f Makefile
mkdir -p target
cc -o target/main.o -c src/main.c
cargo build
  Compiling c-to-rust v0.1.0 (file:///home/viki/rust_cookbook/chapter8/code/c-to-rust)
note: link against the following native artifacts when linking against this static library

note: the order and any duplication can be significant on some platforms, and so may need to be preserved
note: library: dl

note: library: pthread

note: library: gcc_s

note: library: c

note: library: m

note: library: rt

note: library: util

  Finished debug [unoptimized + debuginfo] target(s) in 0.9 secs
cc -o target/double target/main.o target/debug/libdouble_input.a -Wl,--gc-sections -lpthread
target/double
4 * 2 = 8
```


How it works...

In this recipe, we created an external library in Rust, which can be used by other foreign code. The `lib.rs` file inside the `src` folder is the entry point for packages and libraries in Rust.

In `Cargo.toml`, while setting up the project, we set the `crate-type` field as `["staticlib"]`, which lets Cargo know the project should be compiled as a library.



The `lib` section helps when building a specific target. In our case, it is to build a package. The `name` field of a target is the name of the library that will be generated and it is the default name of the package or project. The dashes in the name of the project will be replaced with underscores.

In `lib.rs`, we created a `staticlib` type crate where we have the `double_input` function, which takes an integer input and returns an integer output by multiplying the input by 2. We set the `double_input` function as public using the `pub` keyword, which allows external sources to call the function and the `extern` keyword makes this function stick to the C calling conventions. The `no_mangle` attribute turns off Rust's name mangling so that it is easier to link.

In the `main.c` file, we call the externally created `double_input` function using the `extern` keyword with the declaration of `int32_t double_input(int32_t input)`.

In order to compile and run this program, we used the `make` tool and created some rules, which are sequential steps for executing the project. We created `Makefile` where we first determined the shell using the command `($(shell uname), Darwin)` for detecting the OS type for setting the corresponding flags. If the OS is `Darwin`, then the `LDFLAGS` would be `-Wl,-dead_strip`; else `LDFLAGS` is set as `-Wl,--gc-sections -lpthread`. These are the flags to the linker (ld). The `all` rule is dependant on the `target/double` rule and it executes the same rule once the rules dependant to `target/double` are satisfied.

The `target` rule creates the target directory using the command `mkdir -p $@`, where `$@` refers to the left of the `:` in the rule. The `target/debug` is dependent on the rules, `target/main.o` and `target/debug/libdouble_input.a`. When `target/main.o` runs, we are basically running `cc -o target/main.o -c src/main.c`, which creates the `main.o` in the `target` directory. The `target/debug/libdouble_input.a` runs `cargo build` command, which creates `libdouble_input.a`. The `target/debug` is run with `cc -o target/debug target/main.o target/debug/libdouble_input.a -Wl,--gc-sections -lpthread`, where the links are created. Finally, the executable `double` is run.

Calling Rust operations from Node.js apps

In this recipe, we will create a Rust function, which can be called from JavaScript, and this technique can be really helpful when the developer wants to improve the performance of a certain unit of their web application.

Getting ready

We will have to install the following system dependencies before we can go ahead and create a Rust unit for the Node.js project. Follow the given steps to download and install the node dependencies package for your project:

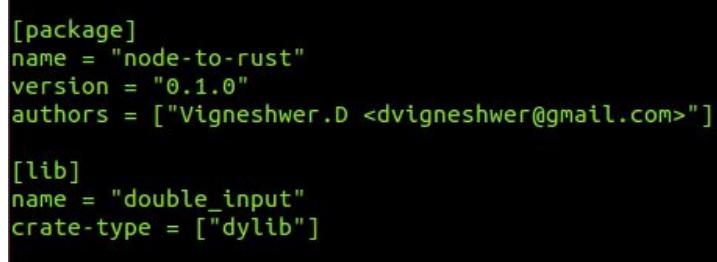
1. We will require the Rust compiler and any text editor to develop the Rust code snippet.
2. Create a Rust library project using the Cargo tool:

```
| cargo new node-to-rust
```

3. Enter the newly created Rust library project:

```
| cd node_to_rust
```

4. Open the `cargo.toml` file in your favorite text editor and make the modification shown in the following screenshot:



```
[package]
name = "node-to-rust"
version = "0.1.0"
authors = ["Vigneshwer.D <dvigneshwer@gmail.com>"]

[lib]
name = "double_input"
crate-type = ["dylib"]
```

5. Install Node.js and `npm` in your machine:

```
| sudo apt-get update
| curl -sL https://deb.nodesource.com/setup_6.x | sudo -E bash -
| sudo apt-get install npm
| sudo apt-get install nodejs
```


How to do it...

Follow the given steps to implement this recipe:

1. Open a file named `lib.rs` located in the `node_to_rust/src` directory and open it your text editor.
2. Write the code header with the relevant information:

```
//-- #####  
//-- Task: Rust Function for Js  
//-- Author: Vigneshwer.D  
//-- Version: 1.0.0  
//-- Date: 14 April 17  
//-- #####
```

3. Create the `double_input` function in the Rust script with the given attributes:

```
#[no_mangle]  
pub extern fn double_input(input: i32) -> i32 {  
    input * 2  
}
```

4. Create the `main.js` file in the `node_to_rust/src` directory, which is the JavaScript code:

```
var ffi = require('ffi');  
  
var lib = ffi.Library('target/debug/libdouble_input', {  
    'double_input': [ 'int', [ 'int' ] ]  
});  
  
var input = 4;  
var output = lib.double_input(input);  
console.log(input + " * 2 = " + output);
```

5. Create `Makefile` in the `node_to_rust/` directory for creating `build` rules:

```
ifeq ($(shell uname), Darwin)  
EXT := dylib  
else  
EXT := so  
endif  
  
all: target/debug/libdouble_input.$(EXT)  
node_modules/ffi  
node src/main.js
```

```
target/debug/libdouble_input.${EXT}: src/lib.rs  
Cargo.toml  
cargo build  
  
node_modules/ffi:  
npm install ffi  
  
clean:  
rm -rf target  
rm -rf node_modules
```

We will get the following output on the successful execution of our code:

```
viki@Vigneshwer:~/rust_cookbook/chapter8/code/node-to-rust$ make -f Makefile  
cargo build  
Compiling node-to-rust v0.1.0 (file:///home/viki/rust_cookbook/chapter8/code/node-to-rust)  
  Finished debug [unoptimized + debuginfo] target(s) in 0.22 secs  
node src/main.js  
4 * 2 = 8
```


How it works...

In this recipe, we create an external library in Rust, which can be used by other foreign code. The `lib.rs` file inside the `src` folder is the entry point for packages and libraries in Rust.

In `Cargo.toml`, while setting up the project, we set the `crate-type` field as `["dylib"]`, which lets Cargo know the project should be compiled as a dynamic library.

In `lib.rs`, we created a `dylib` type crate and we have a `double_input` function, which takes an integer input and returns an integer output by doubling the given input. We can see that it's the `pub` keyword that allows external sources to call the function, and `extern` makes this function stick to the Js calling conventions. The `no_mangle` attribute turns off Rust's name mangling so that it is easier to link.

In the `main.js` file, where we call the externally created `double_input` function using the `ffi` node package, we create a variable, `ffi`, which is loaded with the units of `ffi` node module using the `require` keyword. Using the inbuilt `ffi.library` function, we load the `build` package at `target/debug/libdouble_input` and assign the return object type to the variable `lib`, which contains the method `double_input`. Later, we can use this function as `lib.double_int(input)`, where the `input` variable is assigned to a value of `4` in the previous statements of the Js code.

In order to compile and run this program, we use the `make` tool and create certain rules, which are sequential steps for executing the project. We create `Makefile` where we first determine the `$(shell uname), Darwin` shell for detecting the OS type to set the flags. If the OS is `Darwin`, then `EXT` would be `EXT := dylib;` else `EXT` is set to `so`. The `all` rule runs `target/debug/libdouble_input.$(EXT)` and `node_modules/ffi` before executing `node src/main.js`, which will run the JavaScript and produce the logs. The first rule is `target/debug/libdouble_input.$(EXT)`, which creates the crate by building the

Rust project using the `cargo build` command, and the next rule is `node_modules/ffi`, which installs the node package `ffi` by `npm install ffi`.

Calling Rust operations from Python

In this recipe, we will follow the steps that we implemented for the last two recipes where we created a Rust function and used it in other languages as integral units. Here, we will call the Rust unit using Python.

Getting ready

We will have to install the following system dependencies before we can go ahead and create a Rust unit for the Python project. Follow the given steps to download and install the node dependencies package for your project:

1. We will require the Rust compiler and any text editor to develop the Rust code snippet.
2. Create a Rust library project using the Cargo tool:

```
| cargo new python-to-rust
```

3. Enter the newly created Rust library project:

```
| cd python-to-rust
```

4. Open the `cargo.toml` file in your favorite text editor and make the modification shown in the following screenshot:

```
[package]
name = "python-to-rust"
version = "0.1.0"
authors = ["Vigneshwer.D <dvigneshwer@gmail.com>"]

[lib]
name = "double_input"
crate-type = ["dylib"]
```

5. Install Python on your machine, which is pre-installed on your machine:

```
| sudo apt-get update
| sudo apt-get -y upgrade
```


How to do it...

Follow the given steps to implement this recipe:

1. Create a file named `lib.rs` and open it your text editor.
2. Write the code header with the relevant information:

```
//-- #####  
//-- Task: Rust Function for python  
//-- Author: Vigneshwer.D  
//-- Version: 1.0.0  
//-- Date: 14 April 17  
//-- #####
```

3. Create the `double_input` function in the Rust script with the given attributes:

```
#[no_mangle]  
pub extern fn double_input(input: i32) -> i32 {  
    input * 2  
}
```

4. Create the `main.py` file, which is the Python code:

```
from ctypes import cdll  
from sys import platform  
if platform == 'darwin':  
    prefix = 'lib'  
    ext = 'dylib'  
elif platform == 'win32':  
    prefix = ''  
    ext = 'dll'  
else:  
    prefix = 'lib'  
    ext = 'so'  
lib = cdll.LoadLibrary('target/debug/{}double_input'.format(prefix, ext))  
double_input = lib.double_input  
input = 4  
output = double_input(input)  
print('{} * 2 = {}'.format(input, output))
```

5. Create `Makefile` for creating the `build` rules:

```
ifeq ($(shell uname), Darwin)  
EXT := dylib  
else  
EXT := so  
endif
```

```
all: target/debug/libdouble_input.$(EXT)
    python src/main.py

target/debug/libdouble_input.$(EXT): src/lib.rs
Cargo.toml
cargo build

clean:
rm -rf target
```

We will get the following output on the successful execution of our code:

```
viki@Vigneshwer:~/rust_cookbook/chapter8/code/python-to-rust$ make -f Makefile
cargo build
  Compiling python-to-rust v0.1.0 (file:///home/viki/rust_cookbook/chapter8/code/python-to-rust)
    Finished debug [unoptimized + debuginfo] target(s) in 0.25 secs
python src/main.py
4 * 2 = 8
```


How it works...

In this recipe, we created an external library in Rust, which can be used by other foreign code. The `lib.rs` file inside the `src` folder is the entry point for packages and libraries in Rust.

In `Cargo.toml`, while setting up the project, we set the `crate-type` field as `["dylib"]`, which lets Cargo know the project should be compiled as a dynamic library.

In `lib.rs`, we created a `dylib` type crate and we have a `double_input` function, which takes an integer input and returns an integer output by doubling the given input. We can see that it's the `pub` keyword that allows external sources to call the function, and `extern` makes this function stick to the Js calling conventions. The `no_mangle` attribute turns off Rust's name mangling so that it is easier to link.

In the `main.py` file, we call the externally created `double_input` function build using Rust code by using the `cdll` unit of the `ctypes` Python module where we use the `LoadLibrary` function. We pass the location of the `so`(shared object) file generated after compilation of the Rust project to the `LoadLibrary` Python function, which in our case is located at `target/debug/libdouble_input.so`. On successful loading of the `so` file, we assign `double_input` as `lib.double_input` and call it later in the script.

In order to compile and run this program, we use the `make` tool and create certain rules, which are sequential steps for executing the project. We create a `Makefile` where we first determine the `$(shell uname), Darwin` shell for detecting the OS type to set the flags. If the OS is `Darwin` then `EXT` would be `EXT := dylib; else EXT is set as so.` The `all` rule runs `target/debug/libdouble_input.$(EXT)` before running the Python `src/main.py`. The other rule basically builds the Rust project to build the `libdouble_input.so` file, which is used by the Python script to run the corresponding `double_input` function.

Writing a Python module in Rust

In this recipe, we will create a Python module or library in Rust and import the `create` module in the Python script. Rust provides a great list of safe production-level type systems, so the current Python units of a project can be rewritten in Rust to achieve faster processing and a safer application. Here, we create a library in Rust named `example` with a function named `fibo`, which takes in an index value and produces the Fibonacci sequence value at the index.

Getting ready

We will have to install the following system dependencies before we can go ahead and create a Python module using Rust. Follow the given steps to download and install the dependencies package for your project:

1. We will require the Rust compiler and any text editor for developing the Rust code snippet.
2. Create a Rust library project using the Cargo tool:

```
| cargo new python-rust-library
```

3. Enter the newly created Rust library project:

```
| cd python-rust-library
```

4. Open the `cargo.toml` file in your favorite text editor and make the modification shown in the following screenshot:

```
[package]
name = "python-rust-example"
version = "0.1.0"
authors = ["Vigneshwer.D <dvigneshwer@gmail.com>"]

[lib]
name = "example"
crate-type = ["dylib"]

[dependencies.cpython]
git = "https://github.com/dgrunwald/rust-cpython.git"
default-features = false
features = ["python27-sys"]
```

5. We need to have Python installed on the machine for the executing the steps in this recipe. In most of the systems Python comes pre-installed on the machine, if not run the following commands below:

```
| sudo apt-get update
| sudo apt-get -y upgrade
| sudo apt-get install python-pip python-dev build-essential
```


How to do it...

Follow the given steps to implement this recipe:

1. Create a file named `lib.rs` and open it in your text editor.
2. Write the code header with the relevant information:

```
//-- #####  
//-- Task: Rust-python module  
//-- Author: Vigneshwer.D  
//-- Version: 1.0.0  
//-- Date: 14 April 17  
//-- #####
```

3. Import the `cpython` library and the corresponding module:

```
#[macro_use] extern crate cpython;  
  
use cpython::{Python, PyResult};
```

4. Create the `fibo` function in the Rust language, which is an implementation of the Fibonacci sequence:

```
fn fibo(py: Python, n : u64) -> PyResult<u64> {  
    if n < 2 {  
        return Ok(1)  
    }  
    let mut prev1 = 1;  
    let mut prev2 = 1;  
    for _ in 1..n {  
        let new = prev1 + prev2;  
        prev2 = prev1;  
        prev1 = new;  
    }  
    Ok(prev1)  
}
```

5. Create the Python module interface, which exposes the `fibo` function to Python calls:

```
// To build a Python compatible module we need an  
initialiser which expose the public interface  
py_module_initializer!(example, initexample,  
PyInit_example, |py, m| {  
    // Expose the function fibo as `extern "C"`  
    try! (m.add(py, "fibo", py_fn!(py, fibo(rand_int:  
u64))));  
  
    // Initialiser's macro needs a Result<> as return value
```

```
|     Ok(())  
| );
```

6. Create the `test.py` file outside the `src` directory, which is the Python code:

```
| import example  
  
| # Running the Rust module  
| print(example.fibo(4))
```

7. Create `Makefile` for creating build rules:

```
| all: run  
  
| build:  
| cargo build --release  
| cp ./target/release/libexample.so ./example.so  
  
| run: build  
| python test.py  
  
| clean:  
| cargo clean  
| rm ./example.so
```

We will get the following output on the successful execution of our code:

```
viki@Vigneshwer:~/rust_cookbook/chapter8/code/python-rust-library$ make -f Makefile  
cargo build --release  
  Finished release [optimized] target(s) in 0.0 secs  
cp ./target/release/libexample.so ./example.so  
python test.py  
5
```


How it works...

In this recipe, we create an external library in Rust, which can be used by other foreign code. The `lib.rs` file inside the `src` folder is the entry point for packages and libraries in Rust.

In `Cargo.toml`, while setting up the project, we set the `crate-type` field as `["dylib"]`, which lets Cargo know that the project should be compiled as a dynamic library. The name of the library is `example`.

We installed the latest `rust-cpython` crate from git, which makes it possible to execute Python code from Rust and build a module in Rust for Python. The Python module created can be used for both Python 2.7 and 3; for Python 2.7 we have to enable `features = ["python27-sys"]` while building the Rust project for the first time.

In `lib.rs`, we import the `cpython` crate and use the modules `Python` and `PyResult`. Then, we create a Rust function named `fibo`, which has taken in arguments of the `py: Python` and `n : u64` types, and the return type is `PyResult<u64>`. Here the `py` argument is of the `Python` type and the return value is wrapped to be a `PyResult` type, which is an alias to the `Result` type curated for custom made Python operations. Since we have these types in the `rust-cpython` project, we don't have to explicitly handle type conversions from Python to Rust. In the `fibo` function, we implement the sequence.

If the entered `n` value is less than `2`, we return `OK(1)`, and, for all other values greater than `2`, we have two mutable variable `prev1` and `prev2` initially assigned to `1`. We then iterate using a `for` loop till `n`. Inside the `for` loop, we create an immutable variable, `new`, which is the sum of `prev1` and `prev2`. After the sum operation, the `prev1` value is assigned to `prev2` and `new` is assigned to `prev1`. We then return `OK(prev1)` once the loop ends. The next step is to expose the `fibo` function as part of the module; this can be done with the `py_module_initializer!` and `py_fn!` macros. The `py_module_initializer` macro basically creates a Python-compatible module. We need an initializer that exposes the public interface. To expose the `fibo` function, we use the `try!`

macro by calling `m.add(py, "fibo", py_fn!(py, fibo(rand_int: u64)))`, where `m.add` adds the `fibo` function with the `py_fn!` macro defining the Python module. The initializer's macro needs a `Result<>` as the return value. So, we return `Ok(())` at the end of the macro.

In the `test.py` file, we import the `example` module and print the result of the Fibonacci function for an input index `4` using `print(example.fibo(4))`.

In order to compile and run this program, we use the `make` tool and create certain rules, which are sequential steps for executing the project. We create `Makefile` where we have the `all` rule, which depends on `run`. The `run` rule, in turn, depends on `build`. In the `build` step, the Rust projects are built using `cargo build --release`, which creates the external shared object file `libexample.so` in `target/release`. Post building, we copy the created `so` file as `example.so` in the home directory of the project where the `test.py` code is located. We then execute the `run` rule, which basically runs the `test.py` code and prints the result.

Web Development with Rust

We will be covering the following recipes in this chapter:

- Setting up a web server
- Creating endpoints
- Handling JSONRequests
- Building a custom error handler
- Hosting templates

Introduction

Rust has many packages available that allow the developer to spin up a web server with it, and Rust is an ideal choice for a web project where memory, safety, and speed are of a great significance.

In this chapter, we will be playing with the nickel crate in Rust, which is a web application framework in Rust and provides all the fundamental units required for building a web application in Rust. The fundamental units include setting up a web server, handling web requests, and more. We will focus on the different macros and types that the nickel crate provides us for creating a web application in Rust.

Setting up a web server

In this recipe, you will learn the steps through which you can install the nickel crate into your Rust project and learn the commands with which you can start a web server where you can host your web application.

Getting ready

We will have to install the following nickel crate before we can go ahead and spawn a web server. Follow the given steps to download and set up the `nickel` crate for your project:

1. We will require the Rust compiler and any text editor to develop the Rust code snippet.
2. Create a Rust project using the Cargo tool and enter the newly created project:

```
| cargo new --bin nickel-demo && cd nickel-demo  
  
viki@Vigneshwer:~/rust_cookbook/chapter9$ cargo new nickel-demo --bin && cd nickel-demo  
   Created binary (application) `nickel-demo` project  
viki@Vigneshwer:~/rust_cookbook/chapter9/nickel-demo$ ls  
Cargo.toml  src  
viki@Vigneshwer:~/rust_cookbook/chapter9/nickel-demo$ █
```

3. Open the `Cargo.toml` file in your favorite text editor and make the modification shown in the following screenshot. Here, we are using the `nickel` crate, which is entered in the dependencies field:

```
[package]  
name = "nickel-demo"  
version = "0.1.0"  
authors = ["Vigneshwer.D <dvigneshwer@gmail.com>"]  
  
[dependencies]  
nickel = "*"
```

4. Install the crate in your project with the following command:

```
| cargo build
```

```
viki@Vigneshwer:~/rust_cookbook/chapter9/sample-nickel$ cargo build
  Updating registry `https://github.com/rust-lang/crates.io-index`
  Downloading mime v0.2.4
    Compiling unicode-normalization v0.1.4
    Compiling matches v0.1.4
    Compiling winapi v0.2.8
    Compiling rustc-serialize v0.3.24
    Compiling unicode-bidi v0.2.5
    Compiling libc v0.2.22
    Compiling language-tags v0.2.2
    Compiling idna v0.1.1
    Compiling time v0.1.37
    Compiling lazy_static v0.1.16
    Compiling httparse v1.2.2
    Compiling regex-syntax v0.3.9
    Compiling semver v0.1.20
    Compiling url v1.4.0
    Compiling memchr v0.1.11
    Compiling aho-corasick v0.5.3
    Compiling utf8-ranges v0.1.3
    Compiling traitobject v0.1.0
    Compiling unsafe-any v0.4.1
    Compiling traitobject v0.0.1
    Compiling groupable v0.2.0
    Compiling rustc_version v0.1.7
    Compiling typeable v0.1.2
    Compiling winapi-build v0.1.1
    Compiling cookie v0.2.5
    Compiling kernel32-sys v0.2.2
    Compiling unicase v1.4.0
    Compiling modifier v0.1.0
    Compiling thread-id v2.0.0
    Compiling log v0.3.7
    Compiling thread_local v0.2.7
    Compiling regex v0.1.80
    Compiling typemap v0.3.3
    Compiling hpack v0.2.0
    Compiling plugin v0.2.6
    Compiling mustache v0.6.3
    Compiling num_cpus v1.4.0
    Compiling mime v0.2.4
    Compiling solicit v0.4.4
    Compiling hyper v0.9.18
    Compiling nickel v0.9.0
  Compiling sample-nickel v0.1.0 (file:///home/viki/rust_cookbook/chapter9/sample-nickel)
  Finished debug [unoptimized + debuginfo] target(s) in 53.80 secs
```


How to do it...

Follow the given steps to implement this recipe:

1. Open the `main.rs` file in the `src` directory in your preferred text editor.
2. Write the code header with the relevant information:

```
//-- #####  
//-- Task: Starting a simple hello world nickel web app  
//-- Author: Vigneshwer.D  
//-- Version: 1.0.0  
//-- Date: 20 April 17  
//-- #####
```

3. Import the installed `nickel` crate by using the `extern` keyword:

```
#[macro_use] extern crate nickel;  
  
use nickel::Nickel;
```

4. Define the `main` function in which we declare the `server` instance:

```
fn main() {  
    let mut server = Nickel::new();  
  
    server.utilize(router! {  
        get "*" => |_req, _res| {  
            "Hello world!"  
        }
    });  
  
    server.listen("127.0.0.1:6767");
}
```

5. Save the file and start the server with the following command from the root directory of the project:

```
| cargo run
```

We will get the following output on the successful execution of our code in the terminal:

```
viki@Vigneshwer:~/rust_cookbook/chapter9/nickel-demo$ cargo run  
   Finished debug [unoptimized + debuginfo] target(s) in 0.0 secs  
     Running `target/debug/nickel-demo`  
Listening on http://127.0.0.1:6767  
Ctrl-C to shutdown server
```

Open your favorite browser and redirect to `http://127.0.0.1:6767/` to get the following output:



How it works...

In this recipe, we created a Rust project named `nickel-demo`, which helps us spawn a web server using the `nickel` web application crate.

Starting at the top, we referenced the external `nickel` crate using the `extern` keyword and loaded all its macros with `#[macro_use]`. The `nickel` crate is the application object and surface that holds all public APIs; it's a struct, which implements all the fundamental methods for performing all the web application tasks. In the `main` function, we first assign `server` instances to a mutable variable and create a new `nickel` application object with `Nickel::new()`, which creates an instance of `nickel` with default error handling.

Next, we set up our endpoint routing for which we use the `router!` macro, which listens at `"**"` and provides a simple message `"Hello world!"`, when a `get` request is demanded by the end user. The `get` method of the `nickel` crate or the server instance registers a handler to be used for a specific `get` request. Handlers are assigned to paths and paths are allowed to contain variables and wildcards; we have `"**"` in our case for handlers, which is a wild card entry and basically returns the same response for the get request of any endpoint. A handler added through this API will be attached to the default router. Double pipe characters represent a closure in Rust; this is the place where our `request` and `response` parameters go, which are `_req` and `_res` in our application. Fundamentally, there are structs that contain the request and response data.

Using the `server.utilize` method, we add the endpoint to the server instance and register the handler, which will be invoked among other handlers before each request. The `server.listen` method listens to the API requests on `127.0.0.1:6767`, where it binds and listens for connections on the given host and port.

Creating endpoints

In this recipe, you will learn the steps through which you can install the nickel crate into your Rust project. You will also learn the commands with which we can create custom endpoints in our web application, which would display different messages on being accessed by an end user of the application.

Getting ready

We will have to install the following nickel crate before we can go ahead and spawn a web server. Follow the given steps to download and set up the `nickel` crate for your project:

1. We will require the Rust compiler and any text editor to develop the Rust code snippet.
2. Create a Rust project using the Cargo tool and enter the newly created project:

```
| cargo new --bin nickel-routing && cd nickel-routing
```

```
viki@Vigneshwer:~/rust_cookbook/chapter9$ cargo new nickel-routing --bin && cd nickel-routing
   Created binary (application) `nickel-routing` project
viki@Vigneshwer:~/rust_cookbook/chapter9/nickel-routing$ ls
Cargo.toml  src
viki@Vigneshwer:~/rust_cookbook/chapter9/nickel-routing$ █
```

3. Open the `Cargo.toml` file in your favorite text editor and make the modification shown in the following screenshot. Here, we are using the `nickel` crate, which is entered in the dependencies field:

```
[package]
name = "nickel-routing"
version = "0.1.0"
authors = ["Vigneshwer.D <dvigneshwer@gmail.com>"]

[dependencies]
nickel = "*"
```

4. Install the crate in your project with the following command:

```
| cargo build
```

This command will install all the dependencies of the `nickel` crate in your Rust project.

How to do it...

Follow these steps to implement this recipe:

1. Open the `main.rs` file in the `src` directory in your preferred text editor.
2. Write the code header with the relevant information:

```
|    //--- #####  
|    //--- Task: Routing using nickel  
|    //--- Author: Vigneshwer.D  
|    //--- Version: 1.0.0  
|    //--- Date: 20 April 17  
|    //--- #####
```

3. Import the installed `nickel` crate by using the `extern` keyword:

```
| #[macro_use] extern crate nickel;  
  
| use nickel::{Nickel, HttpRouter};
```

4. Define the `main` function in which we declare the `server` instance:

```
| fn main() {  
|     let mut server = Nickel::new();  
  
|     server.get("/bar", middleware!("This is the /bar  
| handler"));  
|     server.get("/user/:userid", middleware! { |request|  
|         format!("This is user: {:?}", request.param("userid"))  
|     });  
|     server.get("/a/*/d", middleware!("matches /a/b/d but  
| not /a/b/c/d"));  
|     server.get("/a/**/d", middleware!("This matches /a/b/d  
| and also /a/b/c/d"));  
  
|     server.listen("127.0.0.1:6767");  
| }
```

5. Save the file and start the server with the following command from the root directory of the project:

```
| cargo run
```

We will get the following output on the successful execution of our code in the terminal:

```
viki@Vigneshwer:~/rust_cookbook/chapter9/nickel-routing$ cargo run
    Finished debug [unoptimized + debuginfo] target(s) in 0.0 secs
        Running `target/debug/nickel-routing`
Listening on http://127.0.0.1:6767
Ctrl-C to shutdown server
```

Open your favorite browser and redirect to the following endpoints:

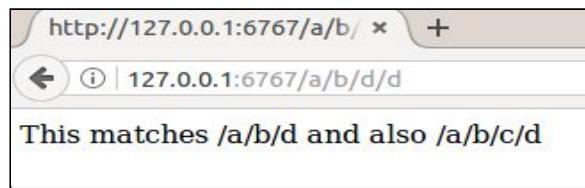
- Enter the URL, `http://127.0.0.1:6767/bar`:



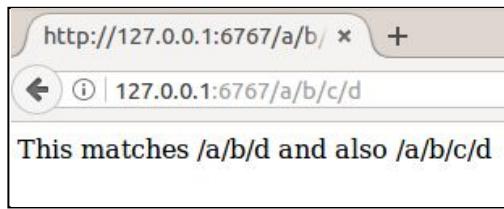
- Enter the URL, `http://127.0.0.1:6767/user/viki`:



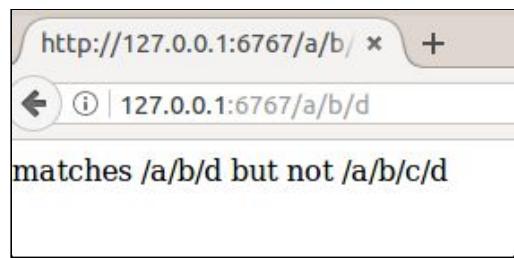
- Enter the URL, `http://127.0.0.1:6767/a/b/d/d`:



- Enter the URL, `http://127.0.0.1:6767/a/b/c/d`:



- Enter the URL, `http://127.0.0.1:6767/a/b/d`:



How it works...

In this recipe, we created a Rust project named `nickel-routing`, which helps us to create multiple endpoints in our web application, and each endpoint displays different custom messages.

Starting at the top, we referenced the external `nickel` crate using the `extern` keyword and loading all of its macros with `#[macro_use]`. We use `nickel` as the application object and surface, which holds all the public APIs. It's a struct that implements all the fundamental methods for performing all the web application tasks, and `HttpRouter` is a public trait provided by the `nickle` crate, which has the signature of various REST API calls.

In the `main` function, we first assign `server` instances to a mutable variable and create a new `nickel` application object with `Nickel::new()`, which creates an instance of `nickel` with default error handling.

The `server.get` method registers a handler to be used for a specific `get` request. Handlers are assigned to paths and paths are allowed to contain variables and wildcards. A handler added through this API will be attached to the default router. The `middleware!` macro reduces the amount of boilerplate code needed for each route.

We create the following routes in this recipe:

- `/bar`: On hitting this endpoint, we get the message, This is the /bar handler.
- `/user/:userid`: On hitting this endpoint, we get the message, This is user: `{:?}`. Here, the argument is replaced with the data (`:userid`) passed in the `get` request with the `request.param("userid")` command, where `param` is a method of the `request` struct.
- `/a/*/d`: On hitting this endpoint, we get the message, matches /a/b/d but not /a/b/c/d. The asterisk here allows only one intermediate path.
- `/a/**/d`: On hitting this endpoint, we get a message: This matches /a/b/d and also /a/b/c/d. The asterisk here allows only two intermediate paths.



Routes can be as simple as `/foo`, use parameters, wildcards, and even double wildcards.

The `server.listen` method listens to the API requests on `127.0.0.1:6767` where it binds and listens for connections on the given host and port.

Handling JSONRequests

In this recipe, we will learn the steps through which we can install the `nickel` crate into your Rust project and learn the commands with which we can accept a `POST` request to an endpoint from the end user.

Getting ready

We will have to install the following `nickel` crate before we can go ahead and spawn a web server. Follow the given steps to download and set up the `nickel` and `rustc_serialize` crates for your project:

1. We will require the Rust compiler and any text editor to develop the Rust code snippet.
2. Create a Rust project using the Cargo tool and enter the newly created project:

```
| cargo new nickel-jsonhandling --bin && cd nickel-jsonhandling
```

Take a look at the following screenshot:

```
viki@Vigneshwer:~/rust_cookbook/chapter9$ cargo new nickel-jsonhandling --bin && cd nickel-jsonhandling
  created binary (application) `nickel-jsonhandling` project
viki@Vigneshwer:~/rust_cookbook/chapter9/nickel-jsonhandling$ ls
Cargo.toml  src
viki@Vigneshwer:~/rust_cookbook/chapter9/nickel-jsonhandling$ █
```

3. Open the `Cargo.toml` file in your favorite text editor and make the modification shown in the following screenshot. Here, we are using the `nickel` and `rustc-serialize` crates, which are entered in the dependencies field:

```
[package]
name = "nickel-jsonhandling"
version = "0.1.0"
authors = ["Vigneshwer.D <vigneshwer.d@mu-sigma.com>"]

[dependencies]
nickel = "*"
rustc-serialize = "0.3.24"
```

4. Install the crate in your project using the following command:

```
| cargo build
```

This command will install all the dependencies of the `nickel` and `rustc-serialize` crate in your Rust project.

How to do it...

Follow the given steps to implement this recipe:

1. Open the `main.rs` file in the `src` directory in your preferred text editor.
2. Write the code header with the relevant information:

```
//-- #####  
//-- Task: Json handling in nickel  
//-- Author: Vigneshwer.D  
//-- Version: 1.0.0  
//-- Date: 20 April 17  
//-- #####
```

3. Import the installed `nickel` and `rustc_serialize` crates by using the `extern` keyword:

```
extern crate rustc_serialize;  
#[macro_use] extern crate nickel;  
  
use nickel::{Nickel, HttpRouter, JsonBody};
```

4. Define a custom `struct` type named `Person`:

```
#[derive(RustcDecodable, RustcEncodable)]  
struct Person {  
    firstname: String,  
    lastname: String,  
}
```

5. Define the `main` function, where we declare the `server` instance:

```
fn main() {  
    let mut server = Nickel::new();  
  
    server.post("/a/post/request", middleware! {  
        |request,  
         response|  
        let person = request.json_as::<Person>().unwrap();  
        format!("Hello {} {}", person.firstname,  
               person.lastname));  
  
        server.listen("127.0.0.1:6767");  
    })
```

6. Save the file and start the server with the following command from the root directory of the project:

```
| cargo run
```

We will get the following output on the successful execution of our code in the terminal:

```
viki@Vigneshwer:~/rust_cookbook/chapter9/code/nickel-jsonhandling$ cargo run
    Finished debug [unoptimized + debuginfo] target(s) in 0.0 secs
     Running `target/debug/nickel-jsonhandling'
Listening on http://127.0.0.1:6767
Ctrl-C to shutdown server
```

Open your terminal and enter the following command to hit the endpoint with `curl`:

```
| curl -H "Content-Type: application/json" -X POST -d
'{"firstname":"Vigneshwer","lastname":"Dhinakaran"}'
http://127.0.0.1:6767/a/post/request
```

On successfully hitting the endpoint with the `curl` command, we will get the following response (highlighted) in the output:

```
viki@Vigneshwer:~/rust_cookbook/chapter9/nickel-jsonhandling$ curl -H
"Content-Type: application/json" -X POST -d '{"firstname":"Vigneshwer","lastname":"Dhinakaran"}'
http://127.0.0.1:6767/a/post/request
Hello Vigneshwer Dhinakaranviki@Vigneshwer:~/rust_cookbook/chapter9/nickel-jsonhandling$
```


How it works...

In this recipe, we created a Rust project named `nickel-jsonhandling`, which helps us get data from the end user and perform a certain set of actions based on the input. The `nickel` crate makes it easy to map JSON data right onto your struct. We use the `rustc-serialize` dependency for this project in order to handle JSON encoding and decoding.

Starting at the top, we referenced the external `nickel` crate using the `extern` keyword and loaded all of its macros with `#[macro_use]`. We use `nickel` as the application object and surface, which holds all the public APIs; it's basically a struct that implements all the fundamental methods for performing the web application tasks. The `HttpRouter` is a public trait provided by the `nickle` crate, which has the signature of various REST API calls. `JsonBody` is a public trait provided by the `nickle` crate, which has the signature of the `json_as` method, which takes a decodable type provided by the `rustc-serialize` crate. We create a custom struct type named `Person`, which has two string fields: `firstname` and `lastname`. The JSON body that is posted from the end user is converted to the `Person` type so that we can use it our application. To be able to encode a piece of data, it must implement the `rustc_serialize::Encodable` trait. To be able to decode a piece of data, it must implement the `rustc_serialize::Decodable` trait. The Rust compiler provides an annotation to automatically generate the code for these traits: `#[derive(RustcDecodable, RustcEncodable)]`.

In the `main` function, we first assign `server` instances to a mutable variable and create a new `nickel` application object with `Nickel::new()`, which creates an instance of `nickel` with default error handling.

The `server.post` method registers a handler to be used for a specific `POST` request. Handlers are assigned to paths and paths are allowed to contain variables and wildcards. A handler added through this API will be attached to the default router. The `middleware!` macro reduces the amount of boilerplate code needed for each route. We create a variable, `person`, which is assigned to `request.json_as::<Person>().unwrap()`, where `request` is a parameter containing

the information from the end user and the `unwrap` method is one of the several ways that Rust provides for assigning a value. We provide a simple message, "Hello {} {}", `person.firstname`, `person.lastname`", in the `format!` macro to be displayed when the `/a/post/request` endpoint is accessed, where `person` is a variable of the `Person` type.

To hit the endpoint, we use the `curl` command, where `-H` stands for the header type, which is "Content-Type: application/json", and we give a POST request (`-x`) at `http://127.0.0.1:6767/a/post/request` with the following data (`-d`):

```
'{"firstname":"Vigneshwer","lastname":"Dhinakaran"}'.
```

The `server.listen` method listens to the API requests on `127.0.0.1:6767` where it binds and listens for connections on the given host and port.

Building custom error handlers

In this recipe, you will learn the steps through which you can install the nickel crate into your Rust project. You will also learn the commands to create your custom error handler, which, for example, can help you create a custom `404` page.

Getting ready

We will have to install the following `nickel` crate before we can go ahead and spawn a web server. Follow the given steps to download and setup a `nickel` crate for your project:

1. We will require the Rust compiler and any text editor to develop the Rust code snippet.
2. Create a Rust project using the Cargo tool and enter the newly created project:

```
| cargo new nickel-errorhandling --bin && cd nickel-errorhandling  
viki@Vigneshwer:~/rust_cookbook/chapter9$ cargo new nickel-errorhandling --bin && cd nickel-errorhandling  
  created binary (application) `nickel-errorhandling` project  
viki@Vigneshwer:~/rust_cookbook/chapter9/nickel-errorhandling$ ls  
cargo.toml  src  
viki@Vigneshwer:~/rust_cookbook/chapter9/nickel-errorhandling$ █
```

3. Open the `cargo.toml` file in your favorite text editor and make the modification shown in the following screenshot. Here, we are using the `nickel` crate, which is entered in the dependencies field:

```
[package]  
name = "nickel-errorhandling"  
version = "0.1.0"  
authors = ["Vigneshwer.D <dvigneshwer@gmail.com>"]  
  
[dependencies]  
nickel = "*"
```

4. Install the crate in your project with the following command:

```
| cargo build
```

This command will install all the dependencies of the `nickle` crate in your Rust project.

How to do it...

Follow the given steps to implement this recipe:

1. Open the `main.rs` file in the `src` directory in your preferred text editor.
2. Write the code header with the relevant information:

```
//-- #####  
//-- Task: Custom error handling in nickel  
//-- Author: Vigneshwer.D  
//-- Version: 1.0.0  
//-- Date: 20 April 17  
//-- #####
```

3. Import the installed `nickel` crate by using the `extern` keyword:

```
#[macro_use] extern crate nickel;  
  
use std::io::Write;  
use nickel::status::StatusCode::NotFound;  
use nickel::{Nickel, NickelError, Action, Continue,  
Halt, Request};
```

4. Define the `main` function in which we declare the `server` instance:

```
fn main() {  
    let mut server = Nickel::new();  
  
    //this is how to overwrite the default error handler  
    to  
    handle 404 cases with a custom view  
    fn custom_404<'a>(err: &mut NickelError, _req: &mut  
Request) -> Action {  
        if let Some(ref mut res) = err.stream {  
            if res.status() == NotFound {  
                let _ = res.write_all(b"<h1>Page Does not exist  
:  
</h1>");  
                return Halt(())  
            }  
        }  
  
        Continue(())
    }  
  
    let custom_handler: fn(&mut NickelError, &mut  
Request) -> Action = custom_404;  
  
    server.handle_error(custom_handler);  
  
    server.listen("127.0.0.1:6767");
}
```

5. Save the file and start the server with the following command from the root directory of the project:

```
| cargo run
```

We will get the following output on the successful execution of our code in the terminal:

```
viki@Vigneshwer:~/rust_cookbook/chapter9/code/nickel-errorhandling$ cargo run
  Finished debug [unoptimized + debuginfo] target(s) in 0.0 secs
    Running `target/debug/nickel-errorhandling'
Listening on http://127.0.0.1:6767
Ctrl-C to shutdown server
```

Open your favorite browser and redirect to `127.0.0.1:6767/viki` to get the following output:



How it works...

In this recipe, we created a Rust project named `nickel-errorhandling`, which helps us spawn the web server using the `nickel` web application crate. By default, `nickel` catches all the errors with its default `ErrorHandler` and tries to take reasonable actions.

Starting at the top, we referenced the external `nickel` crate using the `extern` keyword and loaded all of its macros with `#[macro_use]`.

We used the following units from the `nickel` crate:

- The `nickel` crate is the application object and surface, which holds all the public APIs. It's a struct that implements all the fundamental methods for performing all the web application tasks.
- `NickelError` is a basic error type for `HTTP` errors as well as user-defined errors.
- `Action` is an `enum` data type provided by the `nickel` crate, where `Continue` and `Halt` are variants of the `Action` type.
- `Request` is a container for all the request data.

We used `nickel::status::StatusCode::NotFound`; here, the `status` is a public module of the `nickel` crate, which defines the `StatusCode` enum type containing the different HTTP status codes. `NotFound` is one of them and `std::io::Write`, which is a trait that defines the `write_all` method, writes all the data in the entire buffer.

In the `main` function, we first assign `server` instances to a mutable variable and create a new `nickel` application object with `Nickel::new()`, which creates an instance of `nickel` with default error handling.

We create a custom error handler named `custom_handler`, which is invoked in the case of `NotFound` or `404` status code. We call the function, `custom_404`, which takes in two parameters, `NickelError` and `Request`, and returns an `Action` type.

The `custom_404` is a way to overwrite the default error handler to handle `404` cases with a custom view.

The `custom_404` assigns the arguments `&mut NickelError` and `&mut Request` to `err` and `_req`, respectively. In the function, we assign `err.stream`, where `stream` is a field of the `NickelError` type to `Some(ref mut res)`, and check whether it is true; else we return `Continue(()).` If true, we know that there has been an error when the end user tried to access the endpoint, and the next step is to check whether the status code `res.status()` is `NotFound`. In such a case, we write the custom `404` page with the `res.write_all` method and return `Halt(()).`

The `server.handle_error` registers an error handler, which will be invoked among other error handlers as soon as any regular handler returns an error; the other error handler in our case is `custom_handler`.

The `server.listen` method listens to the API requests on `127.0.0.1:6767` where it binds and listens for connections on the given host and port.

Hosting templates

In this recipe, you will learn the steps through which you can install the `nickel` crate into your Rust project and learn the commands with which we can host a custom template when the end user hits an endpoint.

Getting ready

We will have to install the following `nickel` crate before we can go ahead and spawn a web server. Follow the given steps to download and set up the `nickel` crate for your project:

1. We will require the Rust compiler and any text editor to develop the Rust code snippet.
2. Create a Rust project using the Cargo tool and enter the newly created project:

```
| cargo new nickel-template --bin && cd nickel-template
```

Take a look at the following screenshot:

```
viki@Vigneshwer:~/rust_cookbook/chapter9$ cargo new nickel-template --bin && cd nickel-template
   created binary (application) `nickel-template` project
viki@Vigneshwer:~/rust_cookbook/chapter9/nickel-template$ ls
Cargo.toml  src
viki@Vigneshwer:~/rust_cookbook/chapter9/nickel-template$ █
```

3. Open the `Cargo.toml` file in your favorite text editor and make the modification shown in the following screenshot. Here, we are using the `nickel` crate, which is entered in the dependencies field:

```
[package]
name = "nickel-template"
version = "0.1.0"
authors = ["Vigneshwer.D <vigneshwer.d@mu-sigma.com>"]

[dependencies]
nickel = "*"
```

4. Install the crate in your project with the following command:

```
| cargo build
```

This command will install all the dependencies of the `nickle` crate in your Rust project.

How to do it...

Follow the given steps to implement this recipe:

1. Open the `main.rs` file in the `src` directory in your preferred text editor.
2. Write the code header with the relevant information:

```
//-- #####  
//-- Task: Templating in nickel  
//-- Author: Vigneshwer.D  
//-- Version: 1.0.0  
//-- Date: 20 April 17  
//-- #####
```

3. Import the installed `nickel` crate by using the `extern` keyword:

```
#[macro_use] extern crate nickel;  
  
use std::collections::HashMap;  
use nickel::{Nickel, HttpRouter};
```

4. Define the `main` function where we declare the `server` instance:

```
fn main() {  
    let mut server = Nickel::new();  
  
    server.get("/", middleware! { |_, response|  
        let mut data = HashMap::new();  
        data.insert("name", "viki");  
        return  
        response.render("examples/assets/template.tpl",  
            &data);  
    });  
  
    server.listen("127.0.0.1:6767");  
}
```

5. Create the `examples/assets` directory and the `template.tpl` file using the the following commands:

```
mkdir -p examples/assets  
cd ./examples/assets  
touch template.tpl
```

```
viki@Vigneshwer:~/rust_cookbook/chapter9/nickel-template$ mkdir -p examples/assets  
viki@Vigneshwer:~/rust_cookbook/chapter9/nickel-template$ touch ./examples/assets/template.tpl  
viki@Vigneshwer:~/rust_cookbook/chapter9/nickel-template$ subl ./examples/assets/template.tpl
```

6. Open the file `template.tpl` in a text editor and enter the following code:

```
<html>  
<body>  
<h1>
```

```
Hello {{ name }}!
</h1>
</body>
</html>
```

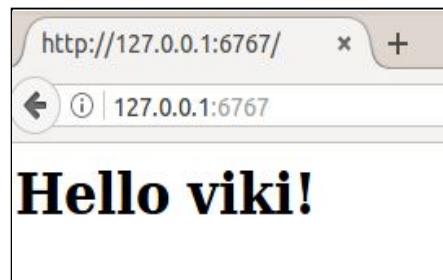
7. Save the file and start the server with the following command from the root directory of the project:

```
| cargo run
```

We will get the following output on the successful execution of our code in the terminal:

```
viki@Vigneshwer:~/rust_cookbook/chapter9/code/nickel-template$ cargo run
  Finished debug [unoptimized + debuginfo] target(s) in 0.0 secs
    Running `target/debug/nickel-template`
Listening on http://127.0.0.1:6767
Ctrl-C to shutdown server
```

Open your favorite browser and redirect to `127.0.0.1:6767` to get the following output:



How it works...

In this recipe, we created a Rust project named `nickel-template`, which loads a custom HTML template with dynamic data fields when the end user tries to access a particular endpoint.

Starting at the top, we referenced the external `nickel` crate using the `extern` keyword and loaded all of its macros with `#[macro_use]`. We used `nickel` as the application object and surface, which holds all the public APIs; it's a struct that implements all the fundamental methods for performing all the web application tasks. We also used `HttpRouter`, which is a public trait provided by the `nickle` crate and has the signature of various REST API calls, and the `HashMap` type from `std::collections`.



HashMaps store values by key. `HashMap` keys can be Booleans, integers, Strings, and vectors. HashMaps are growable but HashMaps can also shrink themselves when they have excess space.

In the `main` function, we first assign `server` instances to a mutable variable and create a new `nickel` application object with `Nickel::new()`, which creates an instance of `nickel` with default error handling.

The `server.get` method registers a handler to be used for a specific `get` request. Handlers are assigned to paths and paths are allowed to contain variables and wildcards. A handler added through this API will be attached to the default router. The `middleware!` macro reduces the amount of boilerplate code needed for each route. The path here is `"/"`, where the response contains the data that needs to be returned to the end user. We host the template created in `examples/assets/template.tpl` and provide the input for the `name` field in the arguments for `response.render`, which is returned as output to the end user. We create a mutable variable named `data`, which is of the `HashMap` type. We also insert a key named `name` and assign it a value, `"viki"` using `data.insert`.

The `server.listen` method listens to the API requests on `127.0.0.1:6767` where it binds and listens for connections on the given host and port.

Advanced Web Development in Rust

We will be covering the following recipes in this chapter:

- Setting up the API
- Saving user data in MongoDB
- Fetching user data
- Deleting user data

Introduction

In this chapter, we will create a RESTful API web service using open source crates in the Rust language. The simple RESTful API in Rust will connect to a MongoDB service, which provides an end-to-end API solution.

We will look at how to perform `GET`, `POST`, and `DELETE` requests on user data from an endpoint. The crates, `nickel.rs` and the MongoDB Rust driver, make it possible to create these actions in the Rust language.

Setting up the API

In this recipe, you will learn the steps through which we can install all the dependencies, such as nickel crate, to set up, and create the API, in our Rust project. You will also learn the commands required to set up the basic REST API service.

Getting ready

We will have to install the following nickel and MongoDB crates before we can go ahead and create the REST API. Follow the steps given to download and set up nickel and the other crate for your project:

1. We will require the Rust compiler and any text editor to develop the Rust code snippet.
2. Create a Rust project using the `cargo` command-line tool, and enter the newly created project:

```
| cargo new --bin sample_rest_api && cd sample_rest_api
```

Take a look at the following screenshot:

```
viki@Vigneshwer:~/rust_cookbook/chapter10$ cargo new --bin sample_rest_api
   Created binary (application) `sample_rest_api` project
viki@Vigneshwer:~/rust_cookbook/chapter10$ cd sample_rest_api/
```

3. Open the `cargo.toml` file in your favorite text editor and make the modifications shown in the following screenshot. Here, we are using the nickel crate, which is entered in the dependencies field:

```
[package]
name = "sample_rest_api"
version = "0.1.0"
authors = ["Vigneshwer.D <dvigneshwer@gmail.com>"]

[dependencies]
nickel = "*"
mongodb = "*"
bson = "*"
rustc-serialize = "*"
```

4. Install the crate in your project with the following command:

```
| cargo build
```

You will get the following screenshot as output:

```
viki@Vigneshwer:~/rust_cookbook/chapter10/sample_rest_api$ cargo build
   Updating registry `https://github.com/rust-lang/crates.io-index`
  Downloading log v0.3.8
  Downloading idna v0.1.2
  Downloading unicode-bidi v0.3.2
  Downloading unsafe-any v0.4.2
  Downloading serde v1.0.8
  Downloading gcc v0.3.49
    Compiling num-traits v0.1.37
    Compiling dtoa v0.4.1
    Compiling libc v0.2.23
    Compiling itoa v0.3.1
    Compiling httparse v1.2.2
    Compiling log v0.3.8
    Compiling modifier v0.1.0
    Compiling scan_fmt v0.1.1
    Compiling mime v0.2.4
    Compiling num-integer v0.1.34
    Compiling winapi v0.2.8
    Compiling groupable v0.2.0
    Compiling rand v0.3.15
    Compiling traitobject v0.1.0
    Compiling lazy_static v0.1.16
    Compiling utf8-ranges v0.1.3
    Compiling unsafe-any v0.4.2
    Compiling linked-hash-map v0.4.2
    Compiling traitobject v0.0.1
    Compiling linked-hash-map v0.3.0
    Compiling byteorder v1.0.0
    Compiling unicode-normalization v0.1.4
```


How to do it...

Perform the following steps to implement this recipe:

1. Open the `main.rs` file in the `src` directory in your preferred text editor.
2. Write the code header with the relevant information:

```
//-- #####  
//-- Task: Creating a simple REST API  
//-- Author: Vigneshwer.D  
//-- Version: 1.0.0  
//-- Date: 28 April 17  
//-- #####
```

3. Import the installed nickel crate by using the `extern` keyword:

```
#[macro_use]  
extern crate nickel;  
  
use nickel::{Nickel, JsonBody, HttpRouter, Request,  
Response, MiddlewareResult, MediaType};
```

4. Define the `main` function in which we declare the `server` instance:

```
fn main() {  
  
    let mut server = Nickel::new();  
    let mut router = Nickel::router();
```

5. Define the `GET` endpoint:

```
router.get("/users", middleware! { |request, response|  
    format!("Hello from GET /users")  
});
```

6. Define the `POST` endpoint:

```
router.post("/users/new", middleware! { |request,  
response|  
    format!("Hello from POST /users/new")  
});
```

7. Define the `DELETE` endpoint:

```
    router.delete("/users/:id", middleware! { |request,
                                             response|
      format!("Hello from DELETE /users/:id")
    });
}
```

8. Declare the port at which the services will be started:

```
server.utilize(router);

server.listen("127.0.0.1:9000");
}
```

9. Save the file and start the server with the following command from the root directory of the project:

```
| cargo run
```

10. We will get the following output on the successful execution of our code in the terminal:

```
viki@Vigneshwer:~/rust_cookbook/chapter10/sample_rest_api$ cargo run
   Finished dev [unoptimized + debuginfo] target(s) in 0.0 secs
     Running `target/debug/sample_rest_api`
Listening on http://127.0.0.1:9000
Ctrl-C to shutdown server
```

We make a `GET` request to `http://127.0.0.1:9000/users`, which returns `Hello from GET /users` from the web service created using the nickel web application framework, as shown in the following screenshot:

The screenshot shows the Postman application interface. At the top, there's a header bar with a URL field containing "http://127.0.0.1:9000/" and a "+" button. To the right are buttons for "No Environment", "Save", and settings. Below the header, there's a toolbar with "GET" selected, a URL input field with "http://127.0.0.1:9000/users", and "Send" and "Save" buttons. A "Code" button is also present. The main area has tabs for "Authorization", "Headers", "Body", "Pre-request Script", and "Tests". Under "Authorization", "Type" is set to "No Auth". The "Body" tab is selected, showing sub-tabs "Body", "Cookies", "Headers (4)", and "Tests". To the right of the body tabs, it says "Status: 200 OK" and "Time: 27 ms". Below the tabs, there are buttons for "Pretty", "Raw", "Preview", and "HTML". The "Preview" tab is active, displaying the response "Hello from GET /users".

We make a `POST` request to `http://127.0.0.1:9000/users/new`, which returns `Hello` from `POST /users/new` from the web service created using the nickel web application framework:

The screenshot shows the Postman interface with a successful `POST` request to `http://127.0.0.1:9000/users/new`. The response status is `200 OK` and the time taken is `24 ms`. The response body contains the text `Hello from POST /users/new`.

We make a `DELETE` request to `http://127.0.0.1:9000/users/:id`, which returns `Hello` from `DELETE /users/:id` from the web service created using the nickel web application framework:

The screenshot shows the Postman interface with a successful `DELETE` request to `http://127.0.0.1:9000/users/:id`. The response status is `200 OK` and the time taken is `23 ms`. The response body contains the text `Hello from DELETE /users/:id`.

How it works...

In this recipe, we created the basic skeleton of our REST API service, where we have all our endpoints set.

Our API will have three endpoints, which are as follows:

- `/users`: Here, we will hit out the `GET` request, which retrieves the `firstname` field of all the users
- `/users/new`: The `POST` request to this endpoint creates and saves a new user
- `/users/:id`: The `DELETE` method will delete a user based on the record's `objectId`, which is the unique ID obtained from the MongoDB database

We will activate all these endpoints in the following recipes with their exact behavior, but currently, these will only display a sample message when the user hits the endpoint with the corresponding requests.

Starting at the top, we're referencing the external nickel crate using the `extern` keyword and loading all of its macros with `# [macro_use]`. The nickel is the application object and the surface that holds all the public APIs; it's a struct, which implements all the fundamental methods for performing all the web application tasks. The other crates, such as `bson`, `rustc-serialize`, and `MongoDB`, which we downloaded initially using the `cargo` tool, will be used in the following recipes to activate the functionalities in the endpoints.

In the `main` function, we first assign the `server` instances to a mutable variable and create a new nickel application object using `Nickel::new()`, which creates an instance of the nickel with default error handling. Similarly, we create a mutable router instance and assign it to `Nickel::router()`, which will take care of handling the different endpoints.

We set up our endpoint routing using different methods, such as `get()`, `post()`, and `delete()`, of the `router` instance and provide a simple message in the

`format!` macro to be displayed when these endpoints are accessed. The messages are as follows:

- `/users`: The `GET` request will get `Hello` from `GET /users`
- `/users/new`: The `POST` request will get `Hello` from `POST /users/new`
- `/users/:id`: The `DELETE` request will get `Hello` from `DELETE /users/:id`

The `middleware!` macro is provided by `nickel.rs` and reduces the amount of boilerplate code needed for each route. Double-pipe characters represent the closure in Rust, and this is where our `request` and `response` parameters are placed.

Using the `server.utilize` method, we add the endpoints to the `server` instance and register the handler that will be invoked among other handlers before each request, by passing the `router` instance. The `server.listen` method listens to the API requests on `127.0.0.1:9000`, where it binds and listens for connections on the given host and port.

Saving user data in MongoDB

In this recipe, we will go a step ahead and rewrite the logic for the `POST` request, which will take the user data and save it in the MongoDB database, for which we will use the MongoDB Rust driver for interacting with the DB.

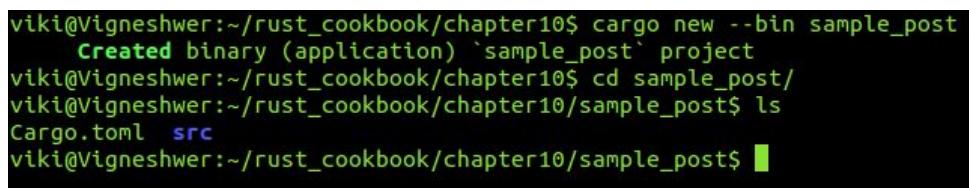
Getting ready

We will have to install the following nickel and MongoDB crates before we can go ahead and create the REST API. Follow these steps to download and set up nickel and the MongoDB service in the project:

1. We will require the Rust compiler and any text editor to develop the Rust code snippet.
2. Create a Rust project using the `cargo` command-line tool and enter the newly created project:

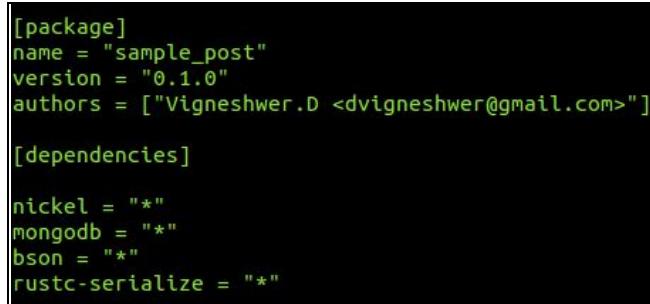
```
| cargo new --bin sample_post && cd sample_post
```

Take a look at the following screenshot:



```
viki@Vigneshwer:~/rust_cookbook/chapter10$ cargo new --bin sample_post
    Created binary (application) `sample_post` project
viki@Vigneshwer:~/rust_cookbook/chapter10$ cd sample_post/
viki@Vigneshwer:~/rust_cookbook/chapter10/sample_post$ ls
Cargo.toml  src
viki@Vigneshwer:~/rust_cookbook/chapter10/sample_post$ █
```

3. Open the `Cargo.toml` file in your favorite text editor and make the modification as shown in the following screenshot. Here, we are using the nickel crate, which is entered in the dependencies field:



```
[package]
name = "sample_post"
version = "0.1.0"
authors = ["Vigneshwer.D <d.vigneshwer@gmail.com>"]

[dependencies]
nickel = "*"
mongodb = "*"
bson = "*"
rustc-serialize = "*"
```

4. Install the crate in your project with the following command:

```
| cargo build
```

5. Set up the MongoDB service in your Linux system by following these steps:

```
sudo apt-key adv --keyserver
hkp://keyserver.ubuntu.com:80 --recv 7F0CEB10

echo "deb http://repo.mongodb.org/apt/ubuntu
"$(lsb_release -sc)"/mongodb-org/3.0 multiverse" | sudo
tee /etc/apt/sources.list.d/mongodb-org-3.0.list

sudo apt-get update

sudo apt-get install -y mongodb-org

service mongod status
```


How to do it...

Follow the mentioned steps to implement this recipe:

1. Open the `main.rs` file in the `src` directory in your preferred text editor.
2. Write the code header with the relevant information:

```
//-- #####  
//-- Task: Perform POST action  
//-- Author: Vigneshwer.D  
//-- Version: 1.0.0  
//-- Date: 28 April 17  
//-- #####
```

3. Import all the required installed nickel and other supporting crates by using the `extern` keyword:

```
#[macro_use]  
extern crate nickel;  
extern crate rustc_serialize;  
  
#[macro_use(bson, doc)]  
extern crate bson;  
extern crate mongodb;  
  
// Nickel  
use nickel::{Nickel, JsonBody, HttpRouter, MediaType};  
use nickel::status::StatusCode::{self};  
  
// MongoDB  
use mongodb::{Client, ThreadedClient};  
use mongodb::db::ThreadedDatabase;  
use mongodb::error::Result as MongoResult;  
  
// bson  
use bson::{Bson, Document};  
use bson::oid::ObjectId;  
  
// rustc_serialize  
use rustc_serialize::json::{Json, ToJson};
```

4. We create a `struct` named `User`, which is encodable and decodable and which models our user data:

```
#[derive(RustcDecodable, RustcEncodable)]  
struct User {  
    firstname: String,  
    lastname: String,  
    email: String  
}
```

5. Define the `main` function, where we'll declare the `server` instance:

```
| fn main() {  
|     let mut server = Nickel::new();  
|     let mut router = Nickel::router();
```

6. Define the `GET` endpoint:

```
|     router.get("/users", middleware! { |request, response|  
|         format!("Hello from GET /users")  
|     });
```

7. Define the `POST` endpoint:

```
|     router.post("/users/new", middleware! { |request,  
|         response|  
  
|         // Accept a JSON string that corresponds to the User  
|         struct  
|         let user = request.json_as::<User>().unwrap();  
  
|         let firstname = user.firstname.to_string();  
|         let lastname = user.lastname.to_string();  
|         let email = user.email.to_string();  
  
|         // Connect to the database  
|         let client = Client::connect("localhost", 27017)  
|             .ok().expect("Error establishing connection.");  
  
|         // The users collection  
|         let coll = client.db("rust-  
|             cookbook").collection("users");  
  
|         // Insert one user  
|         match coll.insert_one(doc! {  
|             "firstname" => firstname,  
|             "lastname" => lastname,  
|             "email" => email  
|         }, None) {  
|             Ok(_) => (StatusCode::Ok, "Item saved!"),  
|             Err(e) => return response.send(format!("{} {}", e))  
|         }  
|     });
```

8. Define the `DELETE` endpoint:

```
|     router.delete("/users/:id", middleware! { |request,  
|         response|  
|         format!("Hello from DELETE /users/:id")  
|     });
```

9. Declare the port at which the services will be started:

```
|     server.utilize(router);  
|  
|     server.listen("127.0.0.1:9000");  
| }
```

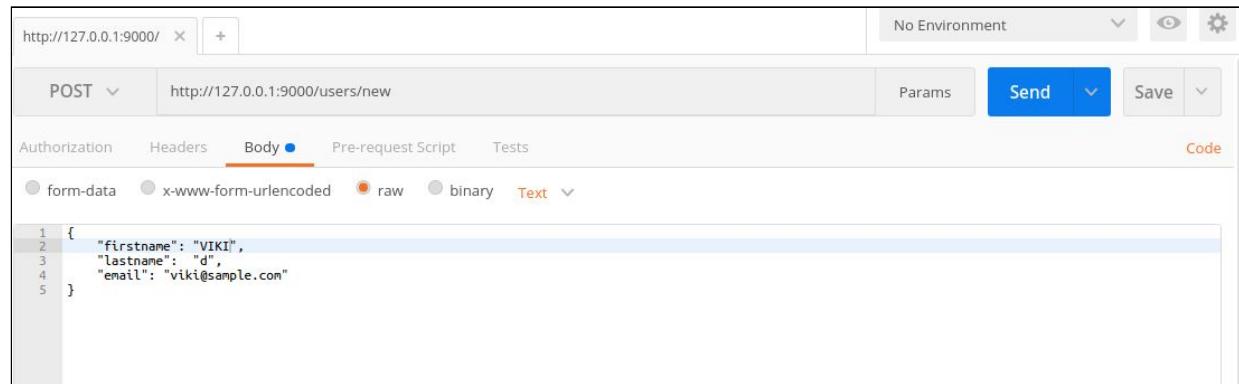
10. Save the file and start the server with the following command from the root directory of the project:

```
| cargo run
```

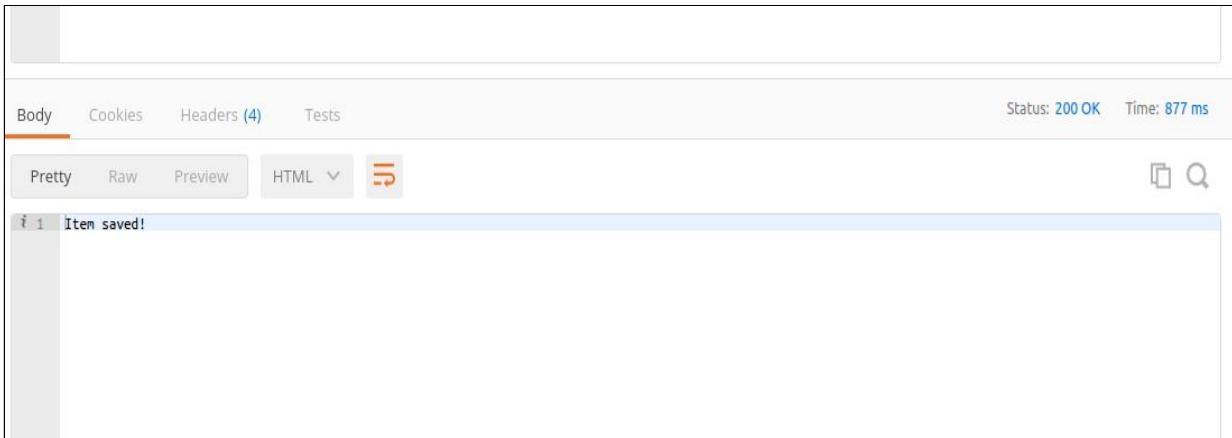
We will get the following screenshot as output on the successful execution of our code in the terminal:

```
viki@Vigneshwer:~/rust_cookbook/chapter10/sample_post$ cargo run  
  Finished dev [unoptimized + debuginfo] target(s) in 0.0 secs  
    Running `target/debug/sample_post`  
Listening on http://127.0.0.1:9000  
Ctrl-C to shutdown server
```

We submit a `POST` request to `http://127.0.0.1:9000/users/new` with a body, as shown in the following screenshot containing the user data:



On a successful API call, the web service built using the nickel web application framework returns a message `Item saved!`, which is an indication of the data being saved in the MongoDB configuration mentioned in the code. Take a look at the following screenshot:



In order to validate whether the `POST` request was successful, we can verify the record from the MongoDB database, where we can select the `rust-cookbook` database and the `users` collection:

```
viki@Vigneshwer:~/rust_cookbook/chapter10/sample_post$ mongo
MongoDB shell version: 3.0.15
connecting to: test
Server has startup warnings:
2017-05-27T06:01:23.440+0530 I CONTROL  [initandlisten]
2017-05-27T06:01:23.440+0530 I CONTROL  [initandlisten] ** WARNING: /sys/kernel/mm/transparent_hugepage/enabled is 'always'.
2017-05-27T06:01:23.440+0530 I CONTROL  [initandlisten] **          We suggest setting it to 'never'
2017-05-27T06:01:23.440+0530 I CONTROL  [initandlisten]
2017-05-27T06:01:23.440+0530 I CONTROL  [initandlisten] ** WARNING: /sys/kernel/mm/transparent_hugepage/defrag is 'always'.
2017-05-27T06:01:23.440+0530 I CONTROL  [initandlisten] **          We suggest setting it to 'never'
2017-05-27T06:01:23.440+0530 I CONTROL  [initandlisten]
> use rust-cookbook
switched to db rust-cookbook
> db.users.find( { "firstname": "VIKI" } ).pretty()
{
    "_id" : ObjectId("592ad247393235202be3f5f6"),
    "firstname" : "VIKI",
    "lastname" : "d",
    "email" : "viki@example.com"
}
> █
```

In this recipe, we focused on getting the user data and saving it to the database. We activated `POST` requests to the `/users/new` endpoint for creating and saving a new user.

Starting at the top, we referenced the external nickel crate using the `extern` keyword and loaded all of its macros with `#[macro_use]`. The nickel is the application object and surface that holds all the public APIs; it's a struct, which implements all the fundamental methods for performing all the web application tasks.

In the `main` function, we first assign `server` mutable instances to a mutable variable and create a new nickel application object with `Nickel::new()`, which creates an instance of nickel with default error handling. Similarly, we create

a mutable router instance and assign it to `Nickel::router()`, which will take care of handling the different endpoints.



The MongoDB Rust driver provides a nice interface for interacting with databases, collections, and cursors, with which we can establish a database connection for creating, reading, updating, and deleting user data in our application.

We will be getting the `POST /users/new` route working, for which we will be using the different crates that we downloaded initially, which are the units of `rustc_serialize`, `bson`, and `MongoDB`.

Next up, we create a complex data structure, which is a `User` struct that is encodable and decodable and that represents our user data fields.

We will need to send a JSON string input from the user end and convert the input data to the `User` struct by creating some variables that can hold the data. The `unwrap` method is one of the several ways that Rust provides for assigning a value. The unwrapped data from the user input is saved to variables `firstname`, `lastname`, and `email`. The next step is to establish a connection with the MongoDB service so that we can store the data that we just parsed from the input string. We achieve this with `Client::connect("localhost", 27017)`, where `27017` is the port in which the MongoDB service is running. The `coll` variable connected the particular collection in the database with `client.db("rust-cookbook").collection("users")`, where `rust-cookbook` is the database and `users` is the collection.

We can see the `match` statement at work when we use `coll.insert_doc` to insert the user data. In the `Ok` condition, we respond with a success message, which is `Item saved`, and in the `Err` condition, we respond with an error.

Using the `server.utilize` method, we add the endpoints to the server instance and register the handler, which will be invoked among other handlers before each request by passing the `router` instance. The `server.listen` method listens to the API requests on `127.0.0.1:9000`, where it binds and listens for connections on the given host and port.

Fetching user data

In this recipe, you will learn the steps with which we can install all the dependencies, such as the nickel crate, to set up and create the API, and the MongoDB Rust driver to interact with the database in our Rust project. You will also learn the commands with which you can get all the specific data required from the database. The `GET` request will fetch the `firstname` field's data, which was previously saved in the database, using the `POST` method.

Getting ready

We will have to install the following nickel and MongoDB crates before we can go ahead and create the REST API. Follow these steps to download and set up nickel and the MongoDB service in the project:

1. We will require the Rust compiler and any text editor to develop the Rust code snippet.
2. Create a Rust project using the `cargo` command-line tool and enter the newly created project:

```
| cargo new --bin sample_get && cd sample_get
```

```
viki@Vigneshwer:~/rust_cookbook/chapter10$ cargo new --bin sample_get && cd sample_get
   Created binary (application) `sample_get` project
viki@Vigneshwer:~/rust_cookbook/chapter10/sample_get$ █
```

3. Open the `cargo.toml` file in your favorite text editor and make the modification as shown in the following screenshot. Here, we are using the nickel crate, which is entered in the dependencies field:

```
[package]
name = "sample_get"
version = "0.1.0"
authors = ["Vigneshwer.D <dvigneshwer@gmail.com>"]

[dependencies]
nickel = "*"
mongodb = "*"
bson = "*"
rustc-serialize = "*"
```

4. Install the crate in your project with the following command:

```
| cargo build
```

5. Set up the MongoDB service in your system by following these steps:

```
sudo apt-key adv --keyserver
hkp://keyserver.ubuntu.com:80 --recv 7F0CEB10

echo "deb http://repo.mongodb.org/apt/ubuntu
$(lsb_release -sc)"/mongodb-org/3.0 multiverse" | sudo
tee /etc/apt/sources.list.d/mongodb-org-3.0.list
```

```
| sudo apt-get update  
| sudo apt-get install -y mongodb-org  
| service mongod status
```


How to do it...

Follow the mentioned steps to implement this recipe:

1. Open the `main.rs` file in the `src` directory in your preferred text editor.
2. Write the code header with the relevant information:

```
//-- #####  
//-- Task: Perform GET action  
//-- Author: Vigneshwer.D  
//-- Version: 1.0.0  
//-- Date: 28 April 17  
//-- #####
```

3. Import all the required installed nickel and other supporting crates by using the `extern` keyword:

```
#[macro_use]  
extern crate nickel;  
extern crate rustc_serialize;  
  
#[macro_use(bson, doc)]  
extern crate bson;  
extern crate mongodb;  
  
// Nickel  
use nickel::{Nickel, JsonBody, HttpRouter, MediaType};  
use nickel::status::StatusCode::{self};  
  
// MongoDB  
use mongodb::{Client, ThreadedClient};  
use mongodb::db::ThreadedDatabase;  
use mongodb::error::Result as MongoResult;  
  
// bson  
use bson::{Bson, Document};  
use bson::oid::ObjectId;  
  
// rustc_serialize  
use rustc_serialize::json::{Json, ToJson};
```

4. We create a `struct` named `User`, which is encodable and decodable and which models our user data:

```
#[derive(RustcDecodable, RustcEncodable)]  
struct User {  
    firstname: String,  
    lastname: String,  
    email: String  
}
```

5. Define the `main` function in which we declare the `server` instance:

```
| fn main() {  
|     let mut server = Nickel::new();  
|     let mut router = Nickel::router();
```

6. Define the `GET` endpoint:

```
| router.get("/users", middleware! { |request, response|  
  
|     // Connect to the database  
|     let client = Client::connect("localhost", 27017)  
|         .ok().expect("Error establishing connection.");  
  
|     // The users collection  
|     let coll = client.db("rust-cookbook").collection("users");  
  
|     // Create cursor that finds all documents  
|     let mut cursor = coll.find(None, None).unwrap();  
  
|     // Opening for the JSON string to be returned  
|     let mut data_result = "{\"data\":[].to_owned()};  
  
|     for (i, result) in cursor.enumerate() {  
  
|         if let Ok(item) = result {  
|             if let Some(&Bson::String(ref firstname)) =  
|                 item.get("firstname") {  
  
|                 let string_data = if i == 0 {  
|                     format!("{}," , firstname)  
|                 } else {  
|                     format!("{}," , firstname)  
|                 };  
|                 data_result.push_str(&string_data);  
|             }  
|         }  
|     }  
  
|     // Close the JSON string  
|     data_result.push_str("]}");  
  
|     // Send back the result  
|     format!("{}", data_result)  
| );
```

7. Define the `POST` endpoint:

```
| router.post("/users/new", middleware! { |request,  
|     response|  
  
|     // Accept a JSON string that corresponds to the User  
|     struct
```

```
let user = request.json_as::<User>().unwrap();

let firstname = user.firstname.to_string();
let lastname = user.lastname.to_string();
let email = user.email.to_string();

// Connect to the database
let client = Client::connect("localhost", 27017)
.ok().expect("Error establishing connection.");

// The users collection
let coll = client.db("rust-cookbook").collection("users");

// Insert one user
match coll.insert_one(doc! {
    "firstname" => firstname,
    "lastname" => lastname,
    "email" => email
}, None) {
    Ok(_) => (StatusCode::Ok, "Item saved!"),
    Err(e) => return response.send(format!("{}", e))
}
});
```

8. Define the `DELETE` endpoint:

```
router.delete("/users/:id", middleware! { |request,
response|  
    format!("Hello from DELETE /users/:id")  
});
```

9. Declare the port at which the services will be started:

```
server.utilize(router);  
  
server.listen("127.0.0.1:9000");  
}
```

10. Save the file and start the server with the following command from the root directory of the project:

```
| cargo run
```

11. We will get the following output on the successful execution of our code in the terminal:

```
viki@Vigneshwer:~/rust_cookbook/chapter10/sample_get$ cargo run
    Finished dev [unoptimized + debuginfo] target(s) in 0.0 secs
      Running `target/debug/sample_get`
Listening on http://127.0.0.1:9000
Ctrl-C to shutdown server
```

We submit a `GET` request to `http://127.0.0.1:9000/users`, as shown in the following screenshot, which fetches the user data on a successful API call, which returns `{"data": [VIKI,]}` from the web service created using the nickel web application framework. The data is fetched from the MongoDB records from the previous recipe where we store this data in the `rust-cookbook` database, as shown in the following screenshot:

The screenshot shows the Postman application interface. At the top, there is a header bar with a URL field containing "http://127.0.0.1:9000/" and a "+" button. To the right are dropdown menus for "No Environment", "Settings", and "Gear". Below the header, there is a toolbar with "GET" dropdown, a search bar with "http://127.0.0.1:9000/users", a "Params" button, a "Send" button, and a "Save" button. Underneath the toolbar, there are tabs for "Authorization", "Headers", "Body", "Pre-request Script", and "Tests". The "Authorization" tab is selected and has a dropdown menu set to "No Auth". Below these tabs, there are buttons for "Body", "Cookies", "Headers (4)", and "Tests". On the far right, it shows "Status: 200 OK" and "Time: 31 ms". At the bottom, there is a preview area with tabs for "Pretty", "Raw", "Preview", and "HTML". The "HTML" tab is selected and displays the JSON response: `i 1 [{"data": [VIKI,]}]`. There are also icons for copy and search at the bottom right of the preview area.

How it works...

In this recipe, we focused on getting previously-saved user data from the database. We activated `GET` requests to the `/users` endpoint for getting the data.

Starting at the top, we referenced the external nickel crate using the `extern` keyword and loaded all of its macros with `#[macro_use]`. The nickel is the application object and surface that holds all the public APIs. It's a struct that implements all the fundamental methods for performing all the web application tasks.

In the `main` function, we first assign `server` mutable instances to a mutable variable and create a new nickel application object with `Nickel::new()`, which creates an instance of nickel with default error handling. Similarly, we create a mutable router instance and assign it to `Nickel::router()`, which will take care of handling the different endpoints.

We will be getting the `GET/users` route working, for which we will be using the different crates that we downloaded initially, which are the units of `rustc_serialize`, `bson`, and `MongoDB`.

Next up, we create a complex data structure, which is a `User` struct that is encodable and decodable and that represents our user data fields.

We first establish a connection with the MongoDB service so that we can store our data, which we just parsed from the input string. We achieve this with `Client::connect("localhost", 27017)`, where `27017` is the port in which the MongoDB service is running, and the `coll` variable connected the particular collection in the database with `client.db("rust-cookbook").collection("users")`, where `rust-cookbook` is the database and `users` is the collection. Then, we have to read through all the values in the collection, for which we create `cursor`, a mutable instance that uses the `find` method to get all the documents in the `users` collection. We create a `data_result` instance, which is a JSON string that will be returned to the user after getting all the data from the collections.

We then iterate over `result` with a `for` loop, where we collect the `i` index and the `Bson` form document with the cursor's `enumerate` method. We convert the value returned, which is of the `Option<Result<Document>` type. We stringify the `firstname` field of the `item` instance using `Bson`, and push the results to the `data_result` string. In the end, we close it off the post, which we send back to the user in the `format!` macro.

Using the `server.utilize` method, we add the endpoints to the server instance and register the handler that will be invoked among other handlers before each request by passing the `router` instance. The `server.listen` method listens to the API requests on `127.0.0.1:9000`, where it binds and listens for connections on the given host and port.

Deleting user data

In this recipe, you will learn the steps with which you can install all the dependencies, such as nickel crate, to set up and create the API, and the MongoDB Rust driver to interact with the database in our Rust project. You will also learn the commands with which you can get all the specific data required from the database. The `GET` request will fetch the `firstname` field's data, which was previously saved in the database by the `POST` method, and then, we will delete the fetched object, which makes this process an end-to-end REST API service.

Getting ready

We will have to install the following nickel and MongoDB crates before we can go ahead and create the REST API. Follow these steps to download and set up nickel and the MongoDB service in the project:

1. We will require the Rust compiler and any text editor to develop the Rust code snippet.
2. Create a Rust project using the `cargo` command-line tool, and enter the newly created project:

```
| cargo new --bin sample_app && cd sample_app
```

```
viki@Vigneshwer:~/rust_cookbook/chapter10$ cargo new --bin sample_app && cd sample_app
   Created binary (application) `sample_app` project
viki@Vigneshwer:~/rust_cookbook/chapter10/sample_app$ █
```

3. Open the `cargo.toml` file in your favorite text editor and make the modification as shown in the following screenshot. Here, we are using the nickel crate, which is entered in the dependencies field:

```
[package]
name = "sample_app"
version = "0.1.0"
authors = ["Vigneshwer.D <dvigneshwer@gmail.com>"]

[dependencies]
nickel = "*"
mongodb = "*"
bson = "*"
rustc-serialize = "*"
```

4. Install the crate in your project with the following command:

```
| cargo build
```

5. Set up the MongoDB service in your system by following these steps:

```
sudo apt-key adv --keyserver
hkp://keyserver.ubuntu.com:80 --recv 7F0CEB10

echo "deb http://repo.mongodb.org/apt/ubuntu
"$(lsb_release -sc)"/mongodb-org/3.0 multiverse" | sudo
tee /etc/apt/sources.list.d/mongodb-org-3.0.list

sudo apt-get update
```

```
| sudo apt-get install -y mongodb-org  
| service mongod status
```


How to do it...

Perform the following steps to implement this recipe:

1. Open the `main.rs` file in the `src` directory in your preferred text editor.
2. Write the code header with the relevant information:

```
///-- #####  
///-- Task: Perform DELETE action  
///-- Author: Vigneshwer.D  
///-- Version: 1.0.0  
///-- Date: 28 April 17  
///-- #####
```

3. Import all the required installed nickel and other supporting crates by using the `extern` keyword:

```
# [macro_use]  
extern crate nickel;  
extern crate rustc_serialize;  
  
# [macro_use(bson, doc)]  
extern crate bson;  
extern crate mongodb;  
  
// Nickel  
use nickel::Nickel, JsonBody, HttpRouter, MediaType;  
use nickel::status::StatusCode::{self};  
  
// MongoDB  
use mongodb::Client, ThreadedClient;  
use mongodb::db::ThreadedDatabase;  
use mongodb::error::Result as MongoResult;  
  
// bson  
use bson::Bson, Document;  
use bson::oid::ObjectId;  
  
// rustc_serialize  
use rustc_serialize::json::{Json, ToJson};
```

4. We create a `struct` named `User`, which is encodable and decodable and which models our user data:

```
# [derive(RustcDecodable, RustcEncodable)]  
struct User {  
    firstname: String,  
    lastname: String,  
    email: String  
}
```

5. Define the `main` function in which we declare the `server` instance:

```
| fn main() {  
|     let mut server = Nickel::new();  
|     let mut router = Nickel::router();
```

6. Define the `GET` endpoint:

```
| router.get("/users", middleware! { |request, response|  
  
|     // Connect to the database  
|     let client = Client::connect("localhost", 27017)  
|         .ok().expect("Error establishing connection.");  
  
|     // The users collection  
|     let coll = client.db("rust-cookbook").collection("users");  
  
|     // Create cursor that finds all documents  
|     let mut cursor = coll.find(None, None).unwrap();  
  
|     // Opening for the JSON string to be returned  
|     let mut data_result = "{\"data\":[".to_owned();  
  
|     for (i, result) in cursor.enumerate() {  
  
|         if let Ok(item) = result {  
|             if let Some(&Bson::String(ref firstname)) =  
|                 item.get("firstname") {  
  
|                 let string_data = if i == 0 {  
|                     format!("{}," , firstname)  
|                 } else {  
|                     format!("{}," , firstname)  
|                 };  
|                 data_result.push_str(&string_data);  
|             }  
|         }  
|     }  
  
|     // Close the JSON string  
|     data_result.push_str("]}");  
  
|     // Send back the result  
|     format!("{}", data_result)  
| );
```

7. Define the `POST` endpoint:

```
| router.post("/users/new", middleware! { |request,  
|     response|  
  
|     // Accept a JSON string that corresponds to the User  
|     struct
```

```

let user = request.json_as::<User>().unwrap();

let firstname = user.firstname.to_string();
let lastname = user.lastname.to_string();
let email = user.email.to_string();

// Connect to the database
let client = Client::connect("localhost", 27017)
.ok().expect("Error establishing connection.");

// The users collection
let coll = client.db("rust-cookbook").collection("users");

// Insert one user
match coll.insert_one(doc! {
    "firstname" => firstname,
    "lastname" => lastname,
    "email" => email
}, None) {
    Ok(_) => (StatusCode::Ok, "Item saved!"),
    Err(e) => return response.send(format!("{}", e))
}
);

});

```

8. Define the `DELETE` endpoint:

```

router.delete("/users/:id", middleware! { |request,
response|

let client = Client::connect("localhost", 27017)
.ok().expect("Failed to initialize standalone
client.");

// The users collection
let coll = client.db("rust-cookbook").collection("users");

// Get the objectId from the request params
let object_id = request.param("id").unwrap();

// Match the user id to an bson ObjectId
let id = match ObjectId::with_string(object_id) {
    Ok(oid) => oid,
    Err(e) => return response.send(format!("{}", e))
};

match coll.delete_one(doc! {"_id" => id}, None) {
    Ok(_) => (StatusCode::Ok, "Item deleted!"),
    Err(e) => return response.send(format!("{}", e))
}

});

```

9. Declare the port at which the services will be started:

```

server.utilize(router);

```

```
|     server.listen("127.0.0.1:9000");  
| }
```

10. Save the file and start the server with the following command from the root directory of the project:

```
| cargo run
```

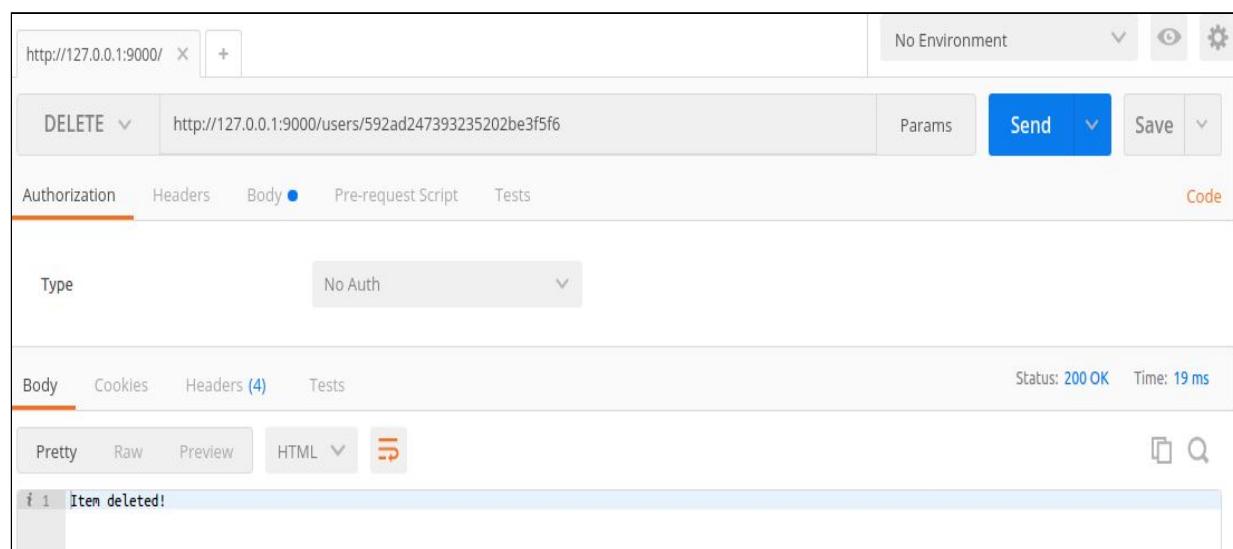
11. We will get the following screenshot as output on the successful execution of our code in the terminal:

```
viki@Vigneshwer:~/rust_cookbook/chapter10/sample_app$ cargo run  
    Finished dev [unoptimized + debuginfo] target(s) in 0.0 secs  
        Running `target/debug/sample_app`  
Listening on http://127.0.0.1:9000  
Ctrl-C to shutdown server
```

We find the `_objectID` of the user data saved, as shown in the following screenshot:

```
> db.users.find()  
{ "_id" : ObjectId("592ad247393235202be3f5f6"), "firstname" : "VIKI", "lastname" : "d", "email" : "viki@example.com" }
```

We submit a `DELETE` request to `http://127.0.0.1:9000/_objectID`, as shown in the following screenshot, which deletes the user data on successful API calls, which returns `Item deleted!` from the web service created using the nickel web application framework, which deleted the data in MongoDB:



How it works...

In this recipe, we focused on deleting the user data that was saved in the database. We activated `DELETE` requests to the `/users/:id` endpoint for deleting the previous records by the object ID.

Starting at the top, we referenced the external nickel crate using the `extern` keyword and loaded all of its macros with `#[macro_use]`. The nickel is the application object and surface that holds all public APIs. It's a struct, which implements all the fundamental methods for performing all the web application tasks.

In the `main` function, we first assign `server` mutable instances to a mutable variable and create a new nickel application object with `Nickel::new()`, which creates an instance of nickel with default error handling. Similarly, we create a mutable router instance and assign it to `Nickel::router()`, which will take care of handling the different endpoints.

We will be getting the `DELETE /users/:id` route working, for which we will be using the different crates that we downloaded initially, which are the units of `rustc_serialize`, `bson`, and `MongoDB`.

Next up, we create a complex data structure, which is a `User` struct that is encodable and decodable and that represents our user data fields.

This is the final step for this end-to-end API, where we allow the `users` collection elements to be deleted by their `objectId`. We can do this with the MongoDB Rust driver's `delete_one` method.

We first establish a connection with the MongoDB service so that we can store our data, which we just parsed from the input string. We achieve this with `Client::connect("localhost", 27017)`, where `27017` is the port in which the MongoDB service is running, and the `coll` variable connected the particular collection in the database with `client.db("rust-cookbook").collection("users")`, where `rust-cookbook` is the database and `users` is the collection.

We get the `objectId` from the request parameters and assign it to `object_id` with `request.param("id").unwrap()`. Then, we use the `ObjectId::with_string` helper to decode the string representation of the `objectId`, after which it can be used in the `delete_one` method to remove the document for that user. With the `DELETE /users/:id` route in place, we should be able to remove `users` from the database when we make a request to it and include `objectId` as a parameter.

Using the `server.utilize` method, we add the endpoints to the server instance and register the handler that will be invoked among other handlers before each request, by passing the `router` instance. The `server.listen` method listens to the API requests on `127.0.0.1:9000`, where it binds and listens for connections on the given host and port.



Consider posting multiple data entries using the `POST` web service to store data values in the MongoDB with different `objectId`, and delete them using the service developed in this recipe to better understand the delete web service working.

Advanced Rust Tools and Libraries

We will cover the following recipes in this chapter:

- Setting up rustup
- Setting up rustfmt
- Setting up rust-clippy
- Setting up and testing with Servo
- Generating random numbers
- Writing lines to a file
- Parsing unstructured JSON
- Parsing URL from a string
- Decompressing a tarball
- Compressing a directory to a tarball
- Finding file extensions recursively

Introduction

In this chapter, you will learn about the various Rust tools, such as `rustfmt` and `rustup`, that help us write better production level Rust code, catch errors, and also provide us with the equipment to do extreme experimentation with different versions of the Rust compiler. Apart from these tools, we will set up and understand the Servo project, which is a state-of-the-art browsing engine that contributes a lot to the design of the Rust language. The last few recipes of this chapter will take the readers through the different ground-level libraries in Rust that contribute a lot to faster project development, by providing the developer with various common operations and functionalities out of the box.

Setting up rustup

In this recipe, you will learn the different options available with the rustup tool that enable us to run the Rust applications in different versions and many more related functionalities.

The rustup tool is the official Rust language installer for rustc, Cargo, and other standard tools to Cargo's `bin` directory. On Unix, it is located at `$HOME/.cargo/bin` and on Windows at `%USERPROFILE%\.cargo\bin`.

Getting ready

We will need to set up the rustup tool in order to perform the different activities of this recipe.

Follow the steps to install the rustup tool:

1. Check for the version of the Rust compiler:

```
|     rustc --version
```

2. If we are able to get an output such as `rustc 1.18.0 (03fc9d622 2017-06-06)`, this means that the rustup tool is set up. The version number may change according to the stable version last downloaded in the system.
3. If you are not able to get the compiler version, follow the installation recipes from [Chapter 1, *Let's Make System Programming Great Again*](#), or run the following command in the Terminal:

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


How to do it...

Follow the steps to implement this recipe:

1. Update the stable version of Rust to the latest version:

```
|   rustup update stable
```

2. Update the `rustup` tool to the latest version:

```
|   rustup self update
```

3. Install the `nightly` toolkit version of the Rust compiler:

```
|   rustup install nightly
```

4. Run the `nightly` version of Rust without changing the default version of Rust:

```
|   rustup run nightly rustc --version
```

5. Change the `default` version of the Rust compiler to the `nightly` version:

```
|   rustup default nightly
```

6. Update the `nightly` version of Rust:

```
|   rustup update
```

7. This will update both the stable and nightly versions of the Rust compiler.

We should get the following output on execution of the `rustup` tool:

- On installation of the nightly version:

```
viki@Vigneshwer:~/rust_cookbook/chapter11$ rustup install nightly
info: syncing channel updates for 'nightly-x86_64-unknown-linux-gnu'
info: latest update on 2017-07-09, rust version 1.20.0-nightly (720c596ec 2017-07-08)
info: downloading component 'rustc'
  38.7 MiB / 38.7 MiB (100 %) 428.8 KiB/s ETA: 0 s
info: downloading component 'rust-std'
  57.9 MiB / 57.9 MiB (100 %) 316.8 KiB/s ETA: 0 s
info: downloading component 'cargo'
  3.7 MiB / 3.7 MiB (100 %) 300.8 KiB/s ETA: 0 s
info: downloading component 'rust-docs'
  3.7 MiB / 3.7 MiB (100 %) 329.6 KiB/s ETA: 0 s
info: downloading component 'rust-std' for 'wasm32-unknown-emscripten'
  12.0 MiB / 12.0 MiB (100 %) 518.4 KiB/s ETA: 0 s
info: downloading component 'rust-std' for 'asmjs-unknown-emscripten'
  12.0 MiB / 12.0 MiB (100 %) 323.2 KiB/s ETA: 0 s
info: removing component 'rustc'
info: removing component 'rust-std'
info: removing component 'cargo'
info: removing component 'rust-docs'
info: removing component 'rust-std' for 'wasm32-unknown-emscripten'
info: removing component 'rust-std' for 'asmjs-unknown-emscripten'
info: installing component 'rustc'
info: installing component 'rust-std'
info: installing component 'cargo'
info: installing component 'rust-docs'
info: installing component 'rust-std' for 'wasm32-unknown-emscripten'
info: installing component 'rust-std' for 'asmjs-unknown-emscripten'

nightly-x86_64-unknown-linux-gnu updated - rustc 1.20.0-nightly (720c596ec 2017-07-08)
```

- Setting nightly as the default version:

```
viki@Vigneshwer:~/rust_cookbook/chapter11$ rustup default nightly
info: using existing install for 'nightly-x86_64-unknown-linux-gnu'
info: default toolchain set to 'nightly-x86_64-unknown-linux-gnu'

nightly-x86_64-unknown-linux-gnu unchanged - rustc 1.20.0-nightly (720c596ec 2017-07-08)
```


How it works...

`rustup` is a toolchain multiplexer that installs and manages many Rust toolchains, and all of them are present in a single set of tools located at `~/.cargo/bin`. As we run examples, `rustup` provides mechanisms to easily change the active toolchain, such as `rustc` and `cargo` at `~/.cargo/bin`, by reconfiguring the locations in the background.

When `rustup` is installed for the first time, the compiler `rustc` will be present in `$HOME/.cargo/bin/rustc`, which, by default, is the stable version. If you later change the default toolchain to nightly with commands such as `rustup install nightly` and `rustup default nightly`, the location will change and the nightly compiler will run instead of the stable one.

The `rustup update` command basically helps us fetch the latest version of the compiler. Rust is distributed on three different release channels: stable, beta, and nightly. `rustup` is configured to use the stable channel by default, which represents the latest release of Rust and is released every six weeks.

Setting up rustfmt

In this recipe, you will learn the steps by which one can set up and use the `rustfmt` tool in their day-to-day Rust project development. `rustfmt` is a tool for formatting the Rust code according to standard style guidelines of the Rust language. It is an open source project and welcomes contribution from the Rust community members. The style guides are decided by the Rust **request for comments (RFC)** procedure, which is an open discussion about the Rust language changes, such as feature requests, bug fixes, and documentation. This is a very important character trait for a programming language as it gives more authority to the developer community than the company that originated the language.

Getting ready

We will follow the steps to set up the tool:

1. Install the nightly version of the Rust compiler:

```
| rustup install nightly
```

2. Set up the Rust compiler version to the nightly toolchain:

```
| rustup default nightly
```

3. Create a new Rust project using Cargo and install the `rustfmt` tool:

```
| cargo new --bin sample_rust_project  
| cd sample_rust_project  
| cargo install rustfmt-nightly
```

You should get the following output on installing the `rustfmt` tool if everything works fine without any errors:

```
viki@Vigneshwer:~/rust_cookbook/chapter11/sample_rust_project/src$ cargo install --force rustfmt-nightly
  Updating registry `https://github.com/rust-lang/crates.io-index'
  Installing rustfmt-nightly v0.1.8
  Downloading toml v0.4.2
  Downloading strings v0.1.0
  Downloading term v0.4.6
  Downloading serde_derive v1.0.9
  Downloading serde v1.0.9
  Downloading thread_local v0.3.4
  Downloading libc v0.2.26
  Downloading unreachable v1.0.0
  Compiling serde v1.0.9
  Compiling regex-syntax v0.4.1
  Compiling diff v0.1.10
  Compiling rustfmt-nightly v0.1.8
  Compiling unicode-xid v0.0.4
  Compiling utf8-ranges v1.0.0
  Compiling void v1.0.2
  Compiling getopts v0.2.14
  Compiling libc v0.2.26
  Compiling unicode-segmentation v1.2.0
  Compiling dtoa v0.4.1
  Compiling num-traits v0.1.39
  Compiling log v0.3.8
  Compiling lazy_static v0.2.8
  Compiling quote v0.3.15
  Compiling itoa v0.3.1
  Compiling term v0.4.6
  Compiling synom v0.11.3
  Compiling unreachable v1.0.0
  Compiling memchr v1.0.1
  Compiling strings v0.1.0
  Compiling syn v0.11.11
  Compiling thread_local v0.3.4
  Compiling toml v0.4.2
  Compiling serde_json v1.0.2
  Compiling aho-corasick v0.6.3
  Compiling regex v0.2.2
  Compiling serde_derive_internals v0.15.1
  Compiling serde_derive v1.0.9
  Compiling env_logger v0.4.3
  Finished release [optimized] target(s) in 200.88 secs
  Replacing /home/viki/.cargo/bin/rustfmt
  Replacing /home/viki/.cargo/bin/cargo-fmt
```


How to do it...

Follow the steps to implement this recipe:

1. Open the `main.rs` file in the `src` directory in your preferred text editor.
2. Write the code header with the relevant information, but the catch here is that we purposefully impose incorrect indentations:

```
//-- #####  
//-- Task: Testing cargo fmt features  
//-- Author: Vigneshwer.D  
//-- Version: 1.0.0  
//-- Date: 04 May 17  
//-- #####
```

3. Define the `main` function with some random spacing and style:

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

4. Now, run `rustfmt` to fix the style issues of the code:

```
| cargo fmt
```

You should get the following output on successfully executing the preceding code:

```
viki@Vigneshwer:~/rust_cookbook/chapter11/sample_rust_project$ ls  
Cargo.lock Cargo.toml src target  
viki@Vigneshwer:~/rust_cookbook/chapter11/sample_rust_project$ cd src/  
viki@Vigneshwer:~/rust_cookbook/chapter11/sample_rust_project/src$ ls  
main.rs  
viki@Vigneshwer:~/rust_cookbook/chapter11/sample_rust_project/src$ cat main.rs  
//-- #####  
//-- Task: Testing cargo fmt features  
//-- Author: Vigneshwer.D  
//-- Version: 1.0.0  
//-- Date: 04 May 17  
//-- #####  
  
fn main() {  
    println!("Hello, world!");  
}  
viki@Vigneshwer:~/rust_cookbook/chapter11/sample_rust_project/src$ cargo fmt  
Warning: the default write-mode for Rustfmt will soon change to overwrite - this will not leave backups of changed files.  
viki@Vigneshwer:~/rust_cookbook/chapter11/sample_rust_project/src$ ls  
main.rs main.rs.bk  
viki@Vigneshwer:~/rust_cookbook/chapter11/sample_rust_project/src$ cat main.rs  
//-- #####  
//-- Task: Testing cargo fmt features  
//-- Author: Vigneshwer.D  
//-- Version: 1.0.0  
//-- Date: 04 May 17  
//-- #####  
  
fn main() {  
    println!("Hello, world!");  
}
```


How it works...

In this recipe, we basically introduce the `rustfmt` tool that helps us follow the correct style guide implied by the Rust programming language in an automated manner, where the developer can freely use the tool to follow style standards.

From the output of the `main.rs` file, we can see that the wrong indentation in the code was automatically corrected and overwritten in the `main.rs` file with the `cargo fmt` command, and that a backup of the previous code was saved in the `main.rs.bk` file.

There are various modes in which we can run the `rustfmt` tool, as follows:

- `replace`: This is the default selection that overwrites the original files and creates backup files after formatting; `cargo fmt` uses `--write-mode=replace` by default.
- `overwrite`: This option basically changes the original files without creating a backup of the previous code.
- `display`: This option basically prints the formatted files to `stdout`, that shows the changes made in the terminal screen.
- `diff`: This option prints the difference between the original files and formatted files to `stdout`. This will also exit with an error code if there are any differences.
- `checkstyle`: This option will output the lines that need to be corrected as a checkstyle XML file that can be used with tools such as Jenkins.

In order to use these write modes, we need to install the `rustfmt` tool by source, where we clone the main GitHub repository and install from the source with the following code:

```
| cargo install --path
```

This enables us to run formatting commands on any Rust files with various modes, such as the following example:

```
| rustfmt main.rs  
| rustfmt --write-mode=overwrite main.rs
```



It is a good practice to run the `rustfmt` tool before building the project so that we can maintain code standards without much effort. The ideal way to do this is to configure the `rustfmt` tool in your favorite text editor and have the `rustfmt` commands as a part of the build process.

Setting up rust-clippy

In this recipe, you will learn the steps to use the rust-clippy tool. The tool basically helps the developer to catch common mistakes and improves your Rust project code by having a collection of lints. The Clippy tool currently works with the latest Rust nightly for now. Since this is a tool for helping the developer of a library or an application to write better code, it is recommended not to include Clippy as a hard dependency and it is mostly used as an optional dependency.

Getting ready

Follow the steps to install the rust-clippy tool:

1. Change the default compiler to the nightly version:

```
|   rustup default nightly
```

2. Set up rust-clippy as an optional dependency by making a modification in the `Cargo.toml` file:

```
| [dependencies]
|   clippy = {version = "*", optional = true}
|
| [features]
|   default = []
```

3. The `Cargo.toml` file will look similar to the following screenshot:

```
viki@Vigneshwer:~/rust_cookbook/chapter11/sample_rust_project$ cat Cargo.toml
[package]
name = "sample_rust_project"
version = "0.1.0"
authors = ["Vigneshwer.D <dvigneshwer@gmail.com>"]

[dependencies]
clippy = {version = "*", optional = true}

[features]
default = []
```

4. Install Clippy from the Cargo tool:

```
|   cargo install clippy
```

5. Set up Clippy as a compiler plugin, by adding Clippy as a dependency to the `Cargo.toml` file:

```
| [dependencies]
|   clippy = "*"
```

6. The `Cargo.toml` file will look similar to the following screenshot:

```
viki@Vigneshwer:~/rust_cookbook/chapter11/sample_rust_project$ cat Cargo.toml
[package]
name = "sample_rust_project"
version = "0.1.0"
authors = ["Vigneshwer.D <dvigneshwer@gmail.com>"]

[dependencies]
clippy = "*"
```

7. Set up a new project for experimenting with the rust-clippy tool:

```
| cargo new --bin sample_rust_project
```


How to do it...

Follow the steps to implement this recipe:

1. Open the `main.rs` file in the `src` directory in your preferred text editor.
2. Write the code header with the relevant information:

```
|    //--- #####  
|    //--- Task: Testing rust-clippy tool  
|    //--- Author: Vigneshwer.D  
|    //--- Version: 1.0.0  
|    //--- Date: 28 April 17  
|    //--- #####
```

3. Create a `main` function, and copy and paste the code:

```
| fn main() {  
|     let x = Some(1u8);  
|     match x {  
|         Some(y) => println!("{}:{}", y),  
|         _ => ()  
|     }  
| }
```

4. Copy and paste the code snippet above the `main` function at the beginning of the code to use rust-clippy as an optional dependency:

```
| #![cfg_attr(feature="clippy", feature(plugin))]  
| #![cfg_attr(feature="clippy", plugin(clippy))]
```

5. Run Clippy with the command `cargo build --features "clippy"`.

6. For running Clippy as a subcommand, run the following command:

```
| cargo clippy
```

7. Copy paste the code snippet above the `main` function at the beginning of the code to use rust-clippy as a compiler plugin:

```
| #![feature(plugin)]  
| #![plugin(clippy)]
```

8. When we run the `cargo run` command, Cargo will install Clippy and show a warning.

We will get the following output for the preceding three different methods:

- Running Clippy as an optional dependency:

```
viki@Vigneshwer:~/rust_cookbook/chapter11/sample_rust_project$ cargo run
Compiling lazy_static v0.2.8
Compiling dtoa v0.4.1
Compiling quote v0.3.15
Compiling semver-parser v0.7.0
Compiling regex-syntax v0.4.1
Compiling itoa v0.3.1
Compiling matches v0.1.6
Compiling num-traits v0.1.39
Compiling unicode-xid v0.0.4
Compiling unicode-normalization v0.1.5
Compiling either v1.1.0
Compiling serde v1.0.9
Compiling quine-mc-cluskey v0.2.4
Compiling synom v0.11.3
Compiling semver v0.6.0
Compiling itertools v0.6.0
Compiling syn v0.11.11
Compiling serde_json v1.0.2
Compiling toml v0.4.2
Compiling serde_derive_internals v0.15.1
Compiling serde_derive v1.0.9
Compiling cargo_metadata v0.2.1
Compiling clippy_lints v0.0.142
Compiling clippy v0.0.142
Compiling sample_rust_project v0.1.0 (file:///home/viki/rust_cookbook/chapter11/sample_rust_project)
warning: you seem to be trying to use match for destructuring a single pattern. Consider using 'if let'
--> src/main.rs:15:5
15 | /     match x {
16 | |     Some(y) => println!("{}?", y),
17 | |     _ => ()
18 | | }
|____^ help: try this `if let Some(y) = x { $crate::io::println(format_args!("{}({})*"), y)`}
= note: #[warn(single_match)] on by default
= help: for further information visit https://github.com/Manishearth/rust-clippy/wiki#single_match

    Finished dev [unoptimized + debuginfo] target(s) in 76.35 secs
        Running `target/debug/sample_rust_project`
```

- Running Clippy as a Cargo subcommand:

```
viki@Vigneshwer:~/rust_cookbook/chapter11/sample_rust_project$ cargo clippy
Compiling sample_rust_project v0.1.0 (file:///home/viki/rust_cookbook/chapter11/sample_rust_project)
warning: you seem to be trying to use match for destructuring a single pattern. Consider using 'if let'
--> src/main.rs:10:5
10 | /     match x {
11 | |     Some(y) => println!("{}?", y),
12 | |     _ => ()
13 | | }
|____^ help: try this `if let Some(y) = x { $crate::io::println(format_args!("{}({})*"), y)`}
= note: #[warn(single_match)] on by default
= help: for further information visit https://github.com/Manishearth/rust-clippy/wiki#single_match

    Finished dev [unoptimized + debuginfo] target(s) in 0.13 secs
```

- Running Clippy as a compiler plugin:

```
viki@Vigneshwer:~/rust_cookbook/chapter11/sample_rust_project$ cargo build --features "clippy"
Compiling sample_rust_project v0.1.0 (file:///home/viki/rust_cookbook/chapter11/sample_rust_project)
warning: you seem to be trying to use match for destructuring a single pattern. Consider using `if let`  

--> src/main.rs:14:5
14 |     match x {  
15 |         Some(y) => println!("{}?", y),  
16 |         _ => ()  
17 |     }  
|____^ help: try this `if let Some(y) = x { $crate::io::println(format_args!(${$arg}*)) }`  
= note: #[warn(single_match)] on by default  
= help: for further information visit https://github.com/Manishearth/rust-clippy/wiki#single\_match  
Finished dev [unoptimized + debuginfo] target(s) in 0.45 secs
```


How it works...

In this recipe, you learned the methods by which you can use the Clippy tool to catch the different lints in the code and get suggestions to improve them.

When we use Clippy as an optional dependency, we set the optional flag as `true` when mentioning Clippy as a dependency in the `Cargo.toml` file. In the feature section, we have a default set of optional packages. Most of the time, developers will want to use these packages, but they are strictly optional. In addition to manipulations in the configuration file, we have to enable the `#[cfg_attr]` attribute that generally works with `#[cfg_attr(condition, attribute)]`, where it allows us to compile the Rust script `main.rs` if the condition is `true`; that is, if the condition is `true`, it's equivalent to `#[attribute]`, and if the condition is `false`, it does not proceed.

The `cargo clippy` command is the easiest way to run the Clippy tool as it does not require any manipulations in the configuration or code.

In the third way, we add the `clippy` dependency in the `Cargo.toml` file without mentioning it as a feature and loading it as a compiler plugin. We do syntax extension by calling `#![feature(plugin)]` and `#![plugin(clippy)]`.



rustc can load compiler plugins, which are user-provided libraries that extend the compiler's behavior with new syntax extensions, lint checks, and so on. There are around 200 lints and this is a growing number configured with the rust-clippy tool that helps developers write better Rust code.

A good place to use the tool would be to have the `cargo clippy` command in the CI script processes for the project before building the projects, similar to the `rustfmt` tool.

Setting up and testing with Servo

In this recipe, we will set up the Servo browser in our systems and test the performance of a website in it.

Servo is a parallel browser engine project sponsored by Mozilla and written in Rust. The Servo project aims to achieve better parallelism, security, modularity, and performance. In short, Servo is a modern, high-performance browser engine designed for both application and embedded use.

Getting ready

We will need to install the following dependencies in our systems to run the Servo browser:

1. Open the Terminal and install the following packages:

```
| sudo apt install git curl freeglut3-dev autoconf libx11-dev \
| libfreetype6-dev libgl1-mesa-dri libglib2.0-dev xorg-dev \
| gperf g++ build-essential cmake virtualenv python-pip \
| libssl1.0-dev libbz2-dev libosmesa6-dev libxmu6 libxmu-dev \
| libglu1-mesa-dev libgles2-mesa-dev libegl1-mesa-dev libdbus-1-dev
```


How to do it...

Follow the steps to implement the build of the Servo project:

1. Open the Terminal, and clone the Servo project from GitHub:

```
| git clone https://github.com/servo/servo
```

2. Enter the project, and build the browser in the development mode:

```
| cd servo  
| ./mach build --dev
```

3. To ensure that the project is built correctly, run a small test:

```
| ./mach run tests/html/about-mozilla.html
```

4. For benchmarking, performance testing, and other real-world scenarios, add the `--release` flag to create an optimized build:

```
| ./mach build --release  
| ./mach run --release tests/html/about-mozilla.html
```

5. For running a website in the Servo project, use the following code:

```
| ./mach run [url]  
| ex: ./mach run https://www.google.com
```


How it works...

In this recipe, you learned to set up the Servo browser engine and run websites in the engine. Servo is a prototype web browser engine written in the Rust language. It is currently developed on 64-bit macOS, 64-bit Linux, and Android.

Servo is built with Cargo and it uses Mozilla's Mach tools to orchestrate the build and other tasks. We run a URL in Servo with `./mach run`
`https://www.google.com.`



There are lots of Mach commands we can use for building and hacking. You can list them with `./mach --help`.

Generating random numbers

In this recipe, we will use the `rand` crate to generate random numbers within a range provided, which in this recipe is 1 to 10. The `rand` crate is a very important library used in various scientific operations.

Getting ready

Follow the following steps to set up the project and install the dependencies:

1. Create a new binary project named `sample_rand`, and enter the directory:

```
| cargo new --bin sample_rand && cd sample_rand
```

2. Install the `cargo-edit` tool that allows you to add and remove dependencies by modifying your `Cargo.toml`:

```
| cargo install cargo-edit
```

3. Install the `rand` dependency:

```
| cargo add rand
```

4. `cargo add crate_name` automatically adds the latest version of the dependency to the `Cargo.toml` file.

5. Install the dependency by running the following command:

```
| cargo run
```

6. This step will print `Hello world` as output, as we have not yet made any modifications to the source code.

How to do it...

Follow the steps to implement this recipe:

1. Open the `main.rs` file in the `src` directory in your preferred text editor.
2. Write the code header with the relevant information:

```
|    //--- #####  
|    //--- Task: To generate a random number between the range of 0 to 10  
|    //--- Author: Vigneshwer.D  
|    //--- Version: 1.0.0  
|    //--- Date: 04 May 17  
|    //--- #####
```

3. Copy paste the following code snippet to the `main.rs` file after the code header:

```
|    extern crate rand;  
|    use rand::Rng;  
|    fn main() {  
|        let mut rng = rand::thread_rng();  
|        println!("{}", rng.gen_range(0, 10));  
|    }
```

4. Save the file and run the project by following the next command:

```
| cargo run
```

We should get the following output on execution of the code:

```
viki@Vigneshwer:~/rust_cookbook/chapter11/code/sample_rand$ cargo run  
Compiling sample_rand v0.1.0 (file:///home/viki/rust_cookbook/chapter11/code/sample_rand)  
  Finished dev [unoptimized + debuginfo] target(s) in 0.54 secs  
    Running `target/debug/sample_rand`  
2  
viki@Vigneshwer:~/rust_cookbook/chapter11/code/sample_rand$ cargo run  
  Finished dev [unoptimized + debuginfo] target(s) in 0.0 secs  
    Running `target/debug/sample_rand`  
3  
viki@Vigneshwer:~/rust_cookbook/chapter11/code/sample_rand$ █
```


How it works...

In this recipe, we print a random number from a range provided by the developer, which in our case is a value between 0 and 10; we use the `rand` library for performing this option.

We import the `rand` library with the `extern crate` command and use the `rand::rng` module that provides us with the `gen_range` method. The `gen_range` method takes two arguments as input that are lower and higher bound values of the range in which it is supposed to predict a random number. In the case of interchanging the position, it would panic.

Writing lines to a file

In this recipe, we will use the `std` crate, which is the Rust standard library providing various modules and functionalities. We will use the filesystem capabilities of the crate to write a three-line message to a file, and then read it back, one line at a time.

Getting ready

Follow the steps to set up the project and install the dependencies:

1. Create a new binary project named `sample_file`, and enter the directory:

```
| cargo new --bin sample_file && cd sample_file
```

2. Install the `error_chain` crate dependency:

```
| cargo add error-chain
```

3. `cargo add crate_name` automatically adds the latest version of the dependency to the `Cargo.toml` file.

4. Install the dependency by running the following command:

```
| cargo run
```

5. This step will print a `Hello world` as output, as we have not yet made any modifications to the source code.

How to do it...

Follow the steps to implement this recipe:

1. Open the `main.rs` file in the `src` directory in your preferred text editor.
2. Write the code header with the relevant information:

```
//-- #####  
//-- Task: File system experiments  
//-- Author: Vigneshwer.D  
//-- Version: 1.0.0  
//-- Date: 04 May 17  
//-- #####
```

3. Create the `error_chain!` macro to define a custom `Error` and `Result` type, along with automatic conversions from the standard library error types, after the code header:

```
# [macro_use]  
extern crate error_chain;  
use std::fs::File;  
use std::io::{Write, BufReader, BufRead};  
error_chain! {  
    foreign_links {  
        Io(std::io::Error);  
    }  
}
```

4. Define the `run` method and the `quick_main!` macro by copying and pasting the following code snippet with the `error_chain!` macro:

```
fn run() -> Result<()> {  
    let path = "lines.txt";  
    let mut output = File::create(path)?;  
    write!(output, "Rust\nnis\nFun")?;  
    let input = File::open(path)?;  
    let buffered = BufReader::new(input);  
    for line in buffered.lines() {  
        println!("{}", line?);  
    }  
    Ok(())  
}  
quick_main!(run);
```

5. Save the file and run the project by running the command:

```
| cargo run
```

We should get the following output on execution of the code:

```
viki@Vigneshwer:~/rust_cookbook/chapter11/code/sample_file$ cargo run
  Compiling sample_file v0.1.0 (file:///home/viki/rust_cookbook/chapter11/code/sample_file)
    Finished dev [unoptimized + debuginfo] target(s) in 1.0 secs
      Running `target/debug/sample_file`
Rust
[1]
Fun
viki@Vigneshwer:~/rust_cookbook/chapter11/code/sample_file$ ls
```


How it works...

In this recipe, we write a three-line message to a file, `lines.txt`. We assign the string `lines.txt` to a variable named `path`, which we pass as an argument to `File::create(path)` that creates the file, and we assign it to a variable `output`. Using the `write!` macro, we write a string `"Rust\n\nFun"` to the mutable `output` variable.

We read the file, then read back with `File::open(path)` and assign it to the `input` variable. We create a variable named `buffered` that stores the content of the file; we then read each line of the file with the `Lines` iterator created by `BufRead::lines` using a `for` loop and print it.



BufrRead is a trait, and the most common way to get one is from a BufReader that is constructed from some type that implements Read; here, a file. The file is opened for writing with File::create and for reading with File::open.

The `error-chain` crate is a library for consistent and reliable error-handling that makes it easier to take full advantage of Rust's powerful error-handling features, without the overhead of maintaining boilerplate error types and conversions. It implements a strategy for defining your own error types, as well as conversions from others' error types.

The basic pattern we use here has a function named `run()` that produces a `Result` type that acts like a *real* `main` function. We use the `error-chain` crate to make `?` work within `run`. This is using the `error_chain!` macro from the `error-chain` crate to define a custom `Error` and `Result` type, along with automatic conversions from the crate error types. The automatic conversions make the `?` operator work. The `quick_main!` macro generates the actual `main` function and prints out the error if it occurs during the course of execution.

We return `Ok(())` to the `quick_run!` macro to ensure that the program executed successfully without any errors.

Parsing unstructured JSON

In this recipe, we will use the `serde_json` crate that provides various modules and functionalities for serializing and deserializing unstructured JSON. We will use the encoding capabilities to parse JSON to a type of the caller's choice.

Getting ready

Follow the steps to set up the project and install the dependencies:

1. Create a new binary project named `sample_json`, and enter the directory:

```
| cargo new --bin sample_json && cd sample_json
```

2. Install the `error_chain` and `serde_json` crates dependencies:

```
| cargo add error-chain serde_json
```

3. `cargo add crate_name` automatically adds the latest version of the dependency to the `Cargo.toml` file.

4. Install the dependency by running the following command:

```
| cargo run
```

5. This step will print `Hello world` as output, as we have not yet made any modifications to the source code.

How to do it...

Follow the steps to implement this recipe:

1. Open the `main.rs` file in the `src` directory in your preferred text editor.
2. Write the code header with the relevant information:

```
//-- #####  
//-- Task: Encoding experiments  
//-- Author: Vigneshwer.D  
//-- Version: 1.0.0  
//-- Date: 04 May 17  
//-- #####
```

3. Create the `error_chain!` macro to define a custom `Error` and `Result` type, along with automatic conversions from the standard library error types, after the code header:

```
# [macro_use]  
extern crate error_chain;  
error_chain! {  
    foreign_links {  
        Json(serde_json::Error);  
    }  
}
```

4. Define the `run` method and the `quick_main!` macro by copying and pasting the following code snippet with the `error_chain!` macro:

```
# [macro_use]  
extern crate serde_json;  
use serde_json::Value;  
fn run() -> Result<()> {  
    let j = r#"  
        "userid": 103609,  
        "verified": true,  
        "access_privileges": [  
            "user",  
            "admin"  
        ]  
    "#;  
    let parsed: Value = serde_json::from_str(j)?;  
    let expected = json!({  
        "userid": 103609,  
        "verified": true,  
        "access_privileges": [  
            "user",  
            "admin"  
        ]  
    });
```

```
    assert_eq!(parsed, expected);
    Ok(())
}
quick_main!(run);
```

5. Save the file and run the project by running the following next command:

```
| cargo run
```

We should get the following output on execution of the code:

```
viki@Vigneshwer:~/rust_cookbook/chapter11/code/sample_json$ cargo run
Compiling sample_json v0.1.0 (file:///home/viki/rust_cookbook/chapter11/code/sample_json)
Finished dev [unoptimized + debuginfo] target(s) in 1.82 secs
Running `target/debug/sample_json`
```


How it works...

In this recipe, we use the `serde_json` crate that provides us with a `from_str` function that allows the developer to parse `&str` of JSON into a type of the caller's choice. We assign a string format of JSON to the `j` variable, after which we call `serde_json::from_str(j)` and assign the output to a variable named `parsed`. The unstructured JSON is parsed into a universal `serde_json::Value` type that represents any valid JSON data.

We then compare, with the `assert_eq!` macro, the value of the parsed `&str` of JSON with what we expect the parsed value to be. The expected value is declared using the `json!` macro.

The `error-chain` crate is a library for consistent and reliable error handling that makes it easier to take full advantage of Rust's powerful error-handling features, without the overhead of maintaining boilerplate error types and conversions. It implements a strategy for defining your own error types, as well as conversions from others' error types.

The basic pattern we use here has a function named `run()` that produces a `Result` that acts like a *real* `main` function. We use the `error-chain` crate to make `?` work within `run`. This is using the `error_chain!` macro from the `error-chain` to define a custom `Error` and `Result` type, along with automatic conversions from the crate error types. The automatic conversions make the `?` operator work. The `quick_main!` macro generates the actual `main` function and prints out the error if it occurs during the course of execution.

We return `Ok(())` to the `quick_run!` macro to ensure that the program executed successfully without any errors.

Parsing URL from a string

In this recipe, we will use the `url` crate that provides various modules and functionalities for networking capabilities. We will use the `parse` functionality of the crate to take a string input and convert it into the URL format after validation.

Getting ready

Follow the following steps to set up the project and install the dependencies:

1. Create a new binary project named `sample_url`, and enter the directory:

```
| cargo new --bin sample_url && cd sample_url
```

2. Install the `error_chain` and `url` crates dependencies:

```
| cargo add error-chain url
```

3. `cargo add crate_name` automatically adds the latest version of the dependency to the `Cargo.toml` file.

4. Install the dependency by running the following command:

```
| cargo run
```

5. This step will print `Hello world` as output, as we have not yet made any modifications to the source code.

How to do it...

Follow the steps to implement this recipe:

1. Open the `main.rs` file in the `src` directory in your preferred text editor.
2. Write the code header with the relevant information:

```
|    //--- #####  
|    //--- Task: Url experiments  
|    //--- Author: Vigneshwer.D  
|    //--- Version: 1.0.0  
|    //--- Date: 04 May 17  
|    //--- #####
```

3. Create the `error_chain!` macro to define a custom `Error` and `Result` type, along with automatic conversions from the standard library error types, after the code header:

```
|    #[macro_use]  
|    extern crate error_chain;  
|    error_chain! {  
|        foreign_links {  
|            UrlParse(url::ParseError);  
|        }  
|    }
```

4. Define the `run` method and the `quick_main!` macro by copying and pasting the following code snippet with the `error_chain!` macro:

```
|    extern crate url;  
|    use url::Url;  
|    fn run() -> Result<()> {  
|        let s = "https://github.com/rust-lang/rust/issues?labels=E-  
|easy&state=open";  
|        let parsed = Url::parse(s)?;  
|        println!("The path part of the URL is: {}", parsed.path());  
|        Ok(())  
|    }  
|    quick_main!(run);
```

5. Save the file and run the project by running the following command:

```
|    cargo run
```

We should get the following output on execution of the code:

```
viki@Vigneshwer:~/rust_cookbook/chapter11/code/sample_urls$ cargo run
  Compiling sample_url v0.1.0 (file:///home/viki/rust_cookbook/chapter11/code/sample_url)
    Finished dev [unoptimized + debuginfo] target(s) in 0.88 secs
      Running `target/debug/sample_url`
The path part of the URL is: /rust-lang/rust/issues
```


How it works...

In this recipe, we use the `parse` method of the `url` crate to validate and parse a `&str` from the data into a `Url` struct. The input string will be transformed into `Result<Url, ParseError>` on the method's return value.

We create a variable named `s` and assign it to the URL that we want to parse; the value is sent to the method with `url::parse(s)`. Once the URL has been parsed, it can be used with all methods on the URL type. Finally we print the path part of the `parsed` variable that stores the return value of the `parse` method.



The URL in this code parses successfully, but swapping it out for a malformed URL will print a message containing an explanation of what went wrong.

The `error-chain` crate is a library for consistent and reliable error handling that makes it easier to take full advantage of Rust's powerful error-handling features, without the overhead of maintaining boilerplate error types and conversions. It implements a strategy for defining your own error types, as well as conversions from others' error types.

The basic pattern we use here has a function named `run()` that produces a `Result` that acts like a *real* `main` function. We use the `error-chain` crate to make `?` work within `run`. This is using the `error_chain!` macro from `error-chain` to define a custom `Error` and `Result` type, along with automatic conversions from the crate error types. The automatic conversions make the `?` operator work. The `quick_main!` macro generates the actual `main` function and prints out the error if it occurs during the course of execution.

We return `Ok(())` to the `quick_run!` macro to ensure that the program executed successfully without any errors.

Decompressing a tarball

In this recipe, we will use the `tar` and `flate2` crates that provide various modules and functionalities for compression capabilities. We will extract the contents of a tar file named `archive.tar.gz` in the current working directory.

Getting ready

Follow the following steps to set up the project and install the dependencies:

1. Create a new binary project named `sample_decom`, and enter the directory:

```
| cargo new --bin sample_decom && cd sample_decom
```

2. Install the `error_chain`, `tar`, and `flate2` crates dependencies:

```
| cargo add error-chain tar flate2
```

3. `cargo add crate_name` automatically adds the latest version of the dependency to the `Cargo.toml` file.

4. Install the dependency by running the following command:

```
| cargo run
```

5. This step will print `hello world` as output, as we have not yet made any modifications to the source code.

6. Create a sample file, TAR it, and delete it for using it in the recipe:

```
| touch sample_file1.txt && tar -cvzf archive.tar.gz  
sample_file1.txt && rm sample_file1.txt
```


How to do it...

Follow the steps to implement this recipe:

1. Open the `main.rs` file in the `src` directory in your preferred text editor.
2. Write the code header with the relevant information:

```
//-- #####  
//-- Task: tar experiments  
//-- Author: Vigneshwer.D  
//-- Version: 1.0.0  
//-- Date: 04 May 17  
//-- #####
```

3. Create the `error_chain!` macro to define a custom `Error` and `Result` type, along with automatic conversions from the standard library error types, after the code header:

```
# [macro_use]  
extern crate error_chain;  
error_chain! {  
    foreign_links {  
        Io(std::io::Error);  
    }  
}
```

4. Define the `run` method and the `quick_main!` macro by copying and pasting the following code snippet with the `error_chain!` macro:

```
extern crate flate2;  
extern crate tar;  
use std::fs::File;  
use flate2::read::GzDecoder;  
use tar::Archive;  
fn run() -> Result<()> {  
    let path = "archive.tar.gz";  
    // Open a compressed tarball  
    let tar_gz = File::open(path)?;  
    // Decompress it  
    let tar = GzDecoder::new(tar_gz)?;  
    // Load the archive from the tarball  
    let mut archive = Archive::new(tar);  
    // Unpack the archive inside current working directory  
    archive.unpack(".")?  
    Ok(())  
}  
quick_main!(run);
```

5. Save the file and run the project by running the following command:

```
| cargo run
```

We should get the following output on execution of the code:

```
vtki@vigneshwer:~/rust_cookbook/chapter11/code/sample_decom$ ls
Cargo.lock Cargo.toml src target
vtki@vigneshwer:~/rust_cookbook/chapter11/code/sample_decom$ touch sample_file1.txt && tar -cvzf archive.tar.gz sample_file1.txt && rm sample_file1.txt
sample_file1.txt
vtki@vigneshwer:~/rust_cookbook/chapter11/code/sample_decom$ ls
archive.tar.gz Cargo.lock Cargo.toml src target
vtki@vigneshwer:~/rust_cookbook/chapter11/code/sample_decom$ cargo run
  Finished dev [unoptimized + debuginfo] target(s) in 0.0 secs
    Running `target/debug/sample_decom`
vtki@vigneshwer:~/rust_cookbook/chapter11/code/sample_decom$ ls
archive.tar.gz Cargo.lock Cargo.toml sample_file1.txt src target
```


How it works...

In this recipe, we open the file using the `File::open(path)` method and assign it to the `tar_gz` variable; the `path` variable contains the string value, which is the location of the `archive.tar.gz` file. We then decompress the `tar_gz` file using the `flate2` crate with the `flate2::read::GzDecoder::new(tar_gz)` command, and assign its return value to the `tar` variable. We extract the files with the `tar::Archive::unpack` method by first creating a variable `archive` by calling `Archive::new(tar)` which loads the archive, and then we get all the files from a compressed tarball named `archive.tar.gz` located in the current working directory by calling `archive.unpack(".").`.

The `error-chain` crate is a library for consistent and reliable error handling that makes it easier to take full advantage of Rust's powerful error-handling features, without the overhead of maintaining boilerplate error types and conversions. It implements a strategy for defining your own error types, as well as conversions from others' error types.

The basic pattern we use here has a function named `run()` that produces a `Result` that acts like a *real* `main` function. We use `error-chain` crate to make `?` work within `run`. This is using the `error_chain!` macro from the `error-chain` to define a custom `Error` and `Result` type, along with automatic conversions from the crate error types. The automatic conversions make the `?` operator work. The `quick_main!` macro generates the actual `main` function and prints out the error if it occurs during the course of execution.

We return `Ok(())` to the `quick_run!` macro to ensure that the program executed successfully without any errors.

Compressing a directory to a tarball

In this recipe, we will use the `tar` and `flate2` crates that provide various modules and functionalities for compression capabilities. We will compress the contents of a directory which, in this recipe, is the current directory to a TAR file named `archive.tar.gz` in the same directory.

Getting ready

Follow the following steps to set up the project and install the dependencies:

1. Create a new binary project named `sample_com`, and enter the directory:

```
| cargo new --bin sample_com && cd sample_com
```

2. Install the `error_chain`, `tar`, and `flate2` crates dependencies:

```
| cargo add error-chain tar flate2
```

3. `cargo add crate_name` automatically adds the latest version of the dependency to the `Cargo.toml` file.

4. Install the dependency by running the following command:

```
| cargo run
```

5. This step will print `Hello world` as output as we have not yet made any modification to the source code.

How to do it...

Follow the steps to implement this recipe:

1. Open the `main.rs` file in the `src` directory in your preferred text editor.
2. Write the code header with the relevant information:

```
//-- #####  
//-- Task: tar experiments  
//-- Author: Vigneshwer.D  
//-- Version: 1.0.0  
//-- Date: 04 May 17  
//-- #####
```

3. Create the `error_chain!` macro to define a custom `Error` and `Result` type, along with automatic conversions from the standard library error types after the code header:

```
#[macro_use]  
extern crate error_chain;  
error_chain! {  
    foreign_links {  
        Io(std::io::Error);  
    }  
}
```

4. Define the `run` method and the `quick_main!` macro by copy pasting the following code snippet the `error_chain!` macro:

```
extern crate tar;  
extern crate flate2;  
use std::fs::File;  
use flate2::Compression;  
use flate2::write::GzEncoder;  
fn run() -> Result<()> {  
    let tar_gz = File::create("archive.tar.gz")?;  
    let enc = GzEncoder::new(tar_gz, Compression::Default);  
    let mut tar = tar::Builder::new(enc);  
    tar.append_dir_all("./backup", "../sample_com")?;  
    Ok(())  
}  
quick_main!(run);
```

5. Save the file and run the project by following the command:

```
| cargo run
```

We should get the following output on execution of the code:

```
vikki@Vigneshwer:~/rust_cookbook/chapter11/code/sample_com$ ls
Cargo.lock  Cargo.toml  src  target
vikki@Vigneshwer:~/rust_cookbook/chapter11/code/sample_com$ cargo run
    Finished dev [unoptimized + debuginfo] target(s) in 0.0 secs
      Running `target/debug/sample_com`
vikki@Vigneshwer:~/rust_cookbook/chapter11/code/sample_com$ ls
archive.tar.gz  Cargo.lock  Cargo.toml  src  target
vikki@Vigneshwer:~/rust_cookbook/chapter11/code/sample_com$ tar -xzf archive.tar.gz
vikki@Vigneshwer:~/rust_cookbook/chapter11/code/sample_com$ ls
archive.tar.gz  backup  Cargo.lock  Cargo.toml  src  target
```


How it works...

In this recipe, we compress the source code directory into `archive.tar.gz`. We create a file with `File::create("archive.tar.gz")` and assign it to the `tar_gz` variable. We then wrap it using `flate2::write::GzEncoder` and `tar::Builder`. Lastly, we add the contents of the `../sample_com` directory recursively into the archive under `./backup` with `Builder::append_dir_all`.



flate2::write::GzEncoder is responsible for transparently compressing the data prior to writing it to archive.tar.gz.

The `error-chain` crate is a library for consistent and reliable error handling that makes it easier to take full advantage of Rust's powerful error-handling features, without the overhead of maintaining boilerplate error types and conversions. It implements a strategy for defining your own error types, as well as conversions from others' error types.

The basic pattern we use here has a function named `run()` that produces a `Result` that acts like a *real* `main` function. We use the `error-chain` crate to make `?` work within `run`. This is using the `error_chain!` macro from the `error-chain` to define a custom `Error` and `Result` type, along with automatic conversions from the crate error types. The automatic conversions make the `?` operator work. The `quick_main!` macro generates the actual `main` function and prints out the error if it occurs during the course of execution.

We return `Ok(())` to the `quick_run!` macro to ensure that the program executed successfully without any errors.

Finding file extensions recursively

In this recipe, we will use the `glob` crate that provides various modules and functionalities for filesystem capabilities. We will recursively find all PNG files in the current directory.

Getting ready

Follow the following steps to set up the project and install the dependencies:

1. Create a new binary project named `sample_ext`, and enter the directory:

```
| cargo new --bin sample_ext && cd sample_ext
```

2. Install the `error_chain` and `glob` crates dependencies:

```
| cargo add error-chain glob
```

3. `cargo add crate_name` automatically adds the latest version of the dependency to the `Cargo.toml` file.

4. Install the dependency by running the following command:

```
| cargo run
```

5. This step will print `Hello world` as outputs, as we have not yet made any modifications to the source code.

How to do it...

Follow the steps to implement this recipe:

1. Open the `main.rs` file in the `src` directory in your preferred text editor.
2. Write the code header with the relevant information:

```
//-- #####  
//-- Task: glob experiments  
//-- Author: Vigneshwer.D  
//-- Version: 1.0.0  
//-- Date: 04 May 17  
//-- #####
```

3. Create the `error_chain!` macro to define a custom `Error` and `Result` type, along with automatic conversions from the standard library error types, after the code header:

```
#[macro_use]  
extern crate error_chain;  
error_chain! {  
    foreign_links {  
        Glob(glob::GlobError);  
        Pattern(glob::PatternError);  
    }  
}
```

4. Define the `run` method and the `quick_main!` macro by copying and pasting the following code snippet with the `error_chain!` macro:

```
extern crate glob;  
use glob::glob;  
fn run() -> Result<()> {  
    for entry in glob("/**/*.*")? {  
        println!("{} ", entry?.display());  
    }  
    Ok(())  
}  
quick_main!(run);
```

5. Save the file and run the project by running the following command:

```
| cargo run
```

We should get the following output on execution of the code:

```
viki@Vigneshwer:~/rust_cookbook/chapter11/code/sample_ext$ cargo run
    Finished dev [unoptimized + debuginfo] target(s) in 0.0 secs
      Running `target/debug/sample_ext`
src/images/B05117_11_01.png
src/images/B05117_11_02.png
src/images/B05117_11_03.png
```


How it works...

In this recipe, we will recursively find all PNG files in the current directory. We use the iteration with a `for` loop with different entries of the `glob("/**/*.png")` method that returns us all the `.png` file extensions in the current directory. We have a `**` pattern that matches the current directory and all subdirectories.



*We can also use the `**` pattern for any directory, not just the current one. For example, `/media/**/*.png` will match all PNGs in media and its subdirectories. For other file extensions, change `.png` to the desired one.*

The `error-chain` crate is a library for consistent and reliable error handling that makes it easier to take full advantage of Rust's powerful error-handling features, without the overhead of maintaining boilerplate error types and conversions. It implements a strategy for defining your own error types, as well as conversions from others' error types.

The basic pattern we use here has a function named `run()` that produces a `Result` that acts like a *real* `main` function. We use the `error-chain` crate to make `? work within run`. This is using the `error_chain!` macro from the `error-chain` to define a custom `Error` and `Result` type, along with automatic conversions from the crate error types. The automatic conversions make the `?` operator work. The `quick_main!` macro generates the actual `main` function and prints out the error if it occurs during the course of execution.

We return `Ok(())` to the `quick_run!` macro to ensure that the program is executed successfully without any errors.

This book was downloaded from AvaxHome!

Visit my blog for more new books:

www.avxhm.se/blogs/AlenMiler