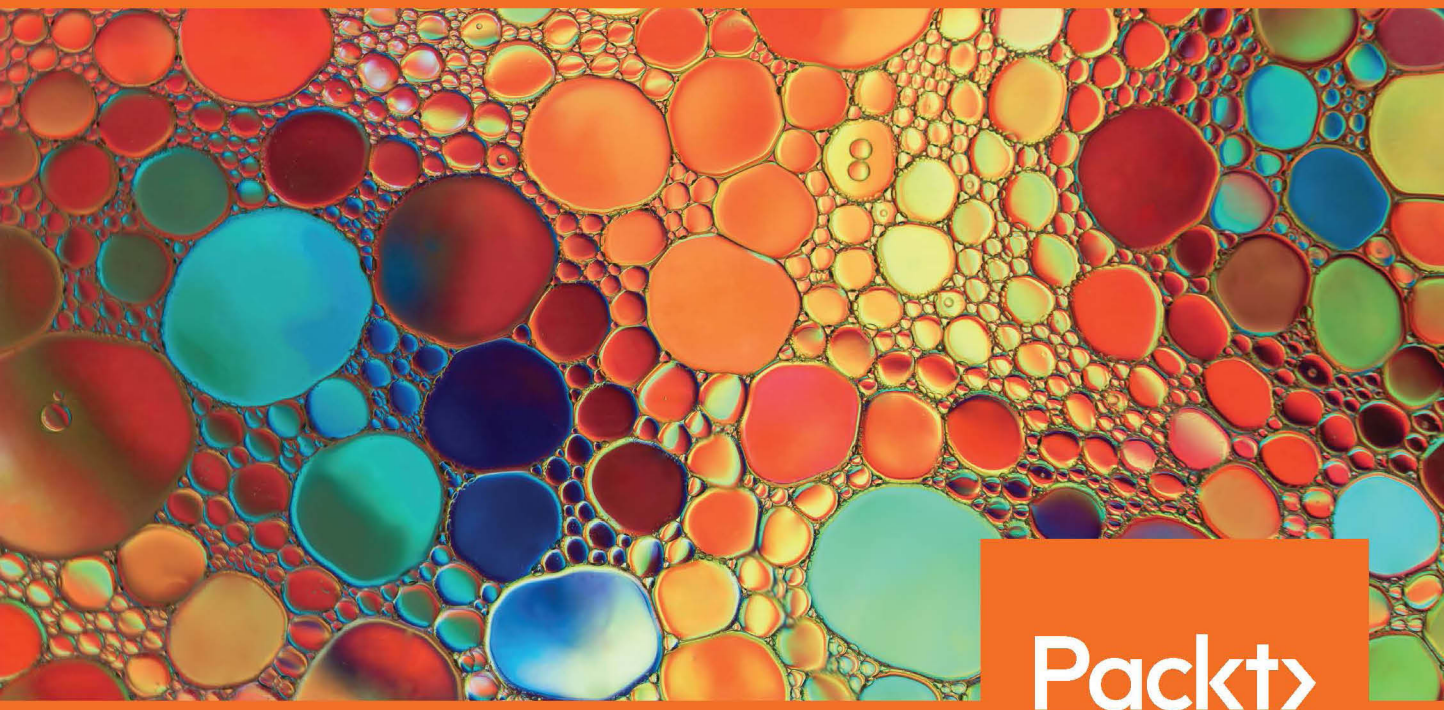
Rust Essentials

Second Edition

A quick guide to writing fast, safe, and concurrent systems and applications



Packt>

www.packt.com

By Ivo Balbaert

Rust Essentials

Second Edition

A quick guide to writing fast, safe, and concurrent systems and applications

Ivo Balbaert



BIRMINGHAM - MUMBAI

Rust Essentials

Second Edition

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: May 2015

Second edition: November 2017

Production reference: 1061117

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham
B3 2PB, UK.
ISBN 978-1-78839-001-9

www.packtpub.com

Credits

Author
Ivo Balbaert

Reviewer
Tom Verbesselt

Commissioning Editor
Merint Mathew

Acquisition Editor
Karan Sadawana

Content Development Editor
Rohit Kumar Singh

Technical Editor
Ruvika Rao

Copy Editor
Safis Editing

Project Coordinator
Vaidehi Sawant

Proofreader
Safis Editing

Indexer
Tejal Daruwale Soni

Graphics
Abhinash Sahu

Production Coordinator
Melwyn Dsa

About the Author

Ivo Balbaert is currently a lecturer in (web) programming and databases at CVO Antwerpen, a community college in Belgium. He received a PhD in applied physics from University of Antwerp in 1986. He has worked for 20 years in the software industry as a developer and consultant in several companies, and he has worked for 10 years as a project manager at the University Hospital of Antwerp. From 2000 onward, he switched to teaching, developing software, and writing technical books.

About the Reviewer

Tom Verbesselt started his career as a software engineer in the amazing early world of 3D scanning and printing. However, he couldn't ignore his passion for teaching, and so he became a lecturer at the graduate program at CVO Antwerpen (Antwerp, Belgium). He started his own IT consultancy firm in 2003 and cofounded OWIC, a non-profit organization dedicated to teaching open source software tools, in 2016. Currently, he is the head of the graduate program in IT at CVO Antwerpen, and he is passionate about mobile, web, and new technology.

> *I have known Ivo for more than 10 years as a colleague at CVO Antwerpen. He is a born teacher and always willing to share his knowledge and insights. He is passionate about programming, and this book will give you all that you need to get started in Rust. This book is a well guided tour around the Rust programming language. I was amazed by the speed, elegance, and possibilities of this programming language. After reading this book, you will have a solid foundation in the language and the advantages and possibilities that it offers. I've loved every page of it, and I hope you will do too.*

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/1788390016`.

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 1

Chapter 1: Starting with Rust 7

- **The advantages of Rust** 8
  - The trifecta of Rust - safe, fast, and concurrent 9
  - Comparison with other languages 11
- **The stability of Rust and its evolution** 12
- **The success of Rust** 13
  - Where to use Rust 15
- **Servo** 15
- **Installing Rust** 16
- **rustc--the Rust compiler** 17
- **Our first program** 17
  - Working with Cargo 19
- **Developer tools** 23
  - Using Sublime Text 24
- **The Standard Library** 26
- **Summary** 26

Chapter 2: Using Variables and Types 27

- **Comments** 28
- **Global constants** 29
  - Printing with string interpolation 30
- **Values and primitive types** 32
  - Consulting Rust documentation 33
- **Binding variables to values** 34
  - Mutable and immutable variables 35
- **Scope of a variable and shadowing** 37
- **Type checking and conversions** 37
  - Aliasing 40
- **Expressions** 40
- **The stack and the heap** 42
- **Summary** 46

Chapter 3: Using Functions and Control Structures 47

- **Branching on a condition** 47

Looping 49
Functions 51
Documenting a function 55
Attributes 56
Conditional compilation 56
Testing 57
Testing with cargo 60
The tests module 61
Summary 63
Chapter 4: Structuring Data and Matching Patterns 64
Strings 65
Arrays, vectors, and slices 69
Vectors 71
Slices 73
Strings and arrays 73
Tuples 75
Structs 77
Enums 79
Result and Option 80
Getting input from the console 81
Matching patterns 84
Program arguments 89
Summary 91
Chapter 5: Higher Order Functions and Error-Handling 92
Higher order functions and closures 92
Iterators 95
Consumers and adapters 97
Generic data structures and functions 99
Error-handling 101
Panics 102
Testing for failure 103
Some more examples of error-handling 104
The try! macro and the ? operator 107
Summary 108
Chapter 6: Using Traits and OOP in Rust 109
Associated functions on structs 110
Methods on structs 112
Using a constructor pattern 113

Using a builder pattern 115
Methods on tuples and enums 116
Traits 117
Using trait constraints 120
Static and dynamic dispatch 123
Built-in traits and operator overloading 125
OOP in Rust 127
Inheritance with traits 127
Using the visitor pattern 129
Summary 131

Chapter 7: Ensuring Memory Safety and Pointers 132

Pointers and references 132
Stack and heap 133
Lifetimes 133
Copying and moving values - The copy trait 137
Let's summarize 141
Pointers 141
References 143
Match, struct, and ref 146
Ownership and borrowing 147
Ownership 147
Moving a value 148
Borrowing a value 148
Implementing the Drop trait 151
Moving closure 152
Boxes 153
Reference counting 157
Overview of pointers 159
Summary 160

Chapter 8: Organizing Code and Macros 161

Modules and crates 162
Building crates 162
Defining a module 164
Visibility of items 165
Importing modules and file hierarchy 167
Importing external crates 169
Exporting a public interface 171
Adding external crates to a project 172
Working with random numbers 174

Macros 176
Why macros? 176
Developing macros 177
Repetition 179
Creating a new function 180
Some other examples 180
Using macros from crates 182
Some other built-in macros 183
Summary 184
Chapter 9: Concurrency - Coding for Multicore Execution 185
Concurrency and threads 185
Creating threads 186
Setting the thread's stack size 188
Starting a number of threads 189
Panicking threads 191
Thread safety 192
Shared mutable states 193
The Sync trait 195
Communication through channels 196
Sending and receiving data 197
Making a channel 198
Sending struct values over a channel 199
Sending references over a channel 199
Synchronous and asynchronous 200
Summary 202
Chapter 10: Programming at the Boundaries 203
When is code unsafe 203
Using std::mem 205
Raw pointers 206
Interfacing with C 207
Using a C library 209
Inlining assembly code 210
Calling Rust from other languages 211
Summary 212
Chapter 11: Exploring the Standard Library 213
Exploring std and the prelude module 213
Collections - using hashmaps and hashsets 214
Working with files 216
Paths 216

Reading a file 218
Error-handling with try! 219
Buffered reading 220
Writing a file 221
Error-handling with try! 223
Filesystem operations 224
Using Rust without the Standard Library 225
Summary 226
Chapter 12: The Ecosystem of Crates 227
The ecosystem of crates 227
Working with dates and times 228
File formats and databases 230
Web development 233
Graphics and games 235
OS and embedded system development 237
Other resources for learning Rust 238
Summary 239
Index 240

Preface

Rust is a stable, open source, and compiled programming language that finally promises software developers the utmost safety--not only type safety, but also memory safety. The compiler carefully checks all uses of variables and pointers so that common problems from C/C++ and other languages, such as pointers to wrong memory locations or null references, are a thing of the past. Possible problems are detected at compilation time, and Rust programs execute at speeds comparable with their C++ counterparts.

Rust runs with a very light runtime, which does not perform garbage collection. Again, the compiler takes care of generating the code that frees all resources at the right time. This means Rust can run in very constrained environments, such as embedded or real-time systems. The built-in safety also guarantees concurrency without data-race problems.

It is clear that Rust is applicable in all use cases where until now C and C++ were the preferred languages and that it will do a better job at it, at least with regard to safety and robustness.

Rust is also a very rich language: it has concepts (such as immutability by default) and constructs (such as traits) that enable developers to write code in a high-level functional and object-oriented style.

The original goal of Rust was to serve as the language for writing a new safe browser engine, devoid of the many security flaws that plague existing browsers such as the Servo project from Mozilla Research.

The goal of this book is to give you a firm foundation for starting to develop in Rust. Throughout the book, we emphasize the three pillars of Rust: safety, performance, and sound concurrency. We will discuss where Rust differs from other programming languages and why this is the case. The code examples are not chosen ad hoc, but they are oriented as part of an ongoing project for building a game so that there is a sense of cohesion and evolution in the examples.

Throughout the book, I will urge you to learn by following along by typing in the code, making the requested modifications, compiling, testing, and working out the exercises.

What this book covers

`Chapter 1`, *Starting with Rust*, discusses the main reasons that led to the development of Rust. We compare Rust with other languages and indicate the areas for which it is most appropriate. Then, we guide you through installing all the necessary components for a Rust development environment. In particular, you will learn how to work with Cargo, Rust's package manager.

`Chapter 2`, *Using Variables and Types*, looks at the basic structure of a Rust program. We discuss the primitive types, how to declare variables and whether they have to be typed, and the scope of variables. Immutability, one of the key cornerstones of Rust's safety strategy is also illustrated. Then, we look at basic operations, how to do formatted printing, and the important difference between expressions and statements.

`Chapter 3`, *Using Functions and Control Structures*, shows how to define functions, and the different ways to influence program execution flow in Rust. We also take a look at attributes and how to do testing in Rust.

`Chapter 4`, *Structuring Data and Matching Patterns*, discusses the basic data types for programming, such as strings, vectors, slices, tuples, and enums. You will learn how to get input from the console and how to work with program arguments. Then we show you the powerful pattern matching that is possible in Rust and how values are extracted by destructuring patterns.

`Chapter 5`, *Higher Order Functions and Error-Handling*, explores the functional features of Rust. We see how data structures and functions can be defined in a generic way. Furthermore, you will learn how to work with Rust's unique error-handling mechanism.

`Chapter 6`, *Using Traits and OOP in Rust*, explores the object-oriented features of Rust. We see how traits can be used to define behavior and to simulate inheritance in data structures. We also explore some common OOP patterns implemented in Rust, such as the visitor and the builder pattern.

`Chapter 7`, *Ensuring Memory Safety and Pointers*, exposes the borrow checker, Rust's mechanism to ensure that only memory safe operations can occur during program execution. Different kinds of pointers are discussed in this chapter.

`Chapter 8`, *Organizing Code and Macros*, discusses the bigger code-organizing structures in Rust, such as modules and crates. It also touches upon how to build macros in order to generate code, thus saving time and effort.

`Chapter 9`, *Concurrency – Coding for Multicore Execution*, delves into Rust's concurrency model based on threads and channels. We also discuss a safe strategy for working with shared mutable data.

`Chapter 10`, *Programming at the Boundaries*, looks at how Rust behaves in situations where we have to leave the safety boundaries, such as when interfacing with C or using raw pointers, and how Rust minimizes potential dangers when doing so.

`Chapter 11`, *Exploring the Standard Library*, gives us an overview of what is contained in Rust's Standard Library with an emphasis on collections and the built-in macros. We also discuss how to let Rust work without standard library, for example, in very resource-constrained environments.

`Chapter 12`, *The Ecosystem of Crates*, covers how to work with crates built by other developers. We look at crates to work with files and databases, do web development, and develop graphics applications and games.

What you need for this book

To run the code examples in the book, you will need the Rust system for your computer, which can be downloaded from `http://www.rust-lang.org/install.html`.

This also contains the Cargo project and package manager. To work more comfortably with Rust code, a development environment like Sublime Text can also be of use. `Chapter 1`, *Starting with Rust*, contains detailed instructions on how to set up your Rust environment.

Who this book is for

This book is directed at developers with some programming experience, either in C / C++, Java/C# or Python, Ruby, Dart or a similar language, having a basic knowledge of general programming concepts. It will get you up and running quickly, giving you all you need to start building your own Rust projects.

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: "The `read_line()` method returns a value of type `IoResult<String>`, which is a specialized `Result` type."

A block of code is set as follows:

```
match magician {
      "Gandalf" => println!("A good magician!"),
      "Sauron"  => println!("A magician turned bad!"),          _
=> println!("No magician turned up!")
   }
```

When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in bold:

```
match magician {
      "Gandalf" => println!("A good magician!"),
      "Sauron"  => println!("A magician turned bad!"),          _
=> println!("No magician turned up!")
   }
```

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

```
# rustc welcome.rs –o start
```

New terms and important words are shown in bold. Words that you see on the screen, for example, in menus or dialog boxes, appear in the text like this: "When working with Rust code, select **Tools | Build System** and **RustEnhanced**."



Warnings or important notes appear in a box like this.



Tips and tricks appear like this.

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**.

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
- Zipeg / iZip / UnRarX for Mac
- 7-Zip / PeaZip for Linux

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

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.

1
Starting with Rust

Rust is a programming language developed at Mozilla Research and backed up by a big open source community. Its development was started in 2006 by language designer Graydon Hoare. Mozilla began sponsoring it in 2009 and it was first presented officially in 2010. Work on this went through a lot of iterations, culminating in early 2015 in the first stable production, version 1.0.0, developed by the Rust Project Developers, consisting of the Rust team at Mozilla and an open source community of over 1800 contributors. Since then, Rust has developed in a steady pace; its current stable version is 1.20.0.

Rust is based on clear and solid principles. It is a systems programming language, equaling C and C++ in its capabilities. It rivals idiomatic C++ in speed, but it lets you work in a much safer way by forbidding code that could cause program crashes due to memory problems. Moreover, it makes concurrent programming and parallel execution on multi-core machines memory safe without garbage collection--it is the only language that does that. By design, Rust eliminates the corruption of shared data through concurrent access, called **data races**.

This chapter will present you with the main reasons why Rust's popularity and adoption are steadily increasing. Then, we'll set up a working Rust development environment.

We will cover the following:

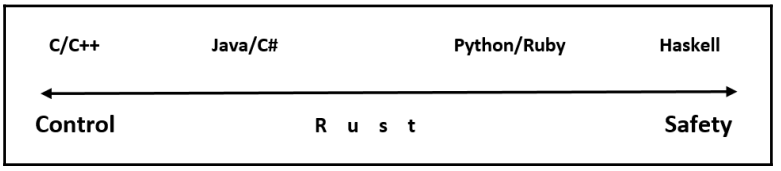
- The advantages of Rust
- The trifecta of Rust--safe, fast and concurrent
- The stability of Rust and its evolution
- The success of Rust
- Using Rust
- Installing Rust
- The Rust compiler
- Our first program

- Working with Cargo
- Developer tools
- The Standard Library

The advantages of Rust

Mozilla is a company known for its mission to develop tools for and drive the open standards web, most notably through its flagship browser Firefox. Every browser today, including Firefox, is written in C++, some 1,29,00,992 lines of code for Firefox, and 44,90,488 lines of code for Chrome. This makes them fast, but it is inherently unsafe because the memory manipulations allowed by C and C++ are not checked for validity. If the code is written without the utmost programming discipline on the part of the developers, program crashes, memory leaks, segmentation faults, buffer overflows, and null pointers can occur at program execution. Some of these can result in serious security vulnerabilities, all too familiar in existing browsers. Rust is designed from the ground up to avoid those kind of problems.

Compared to C or C++, on the other side of the programming language spectrum we have Haskell, which is widely known to be a very safe and reliable language, but with very little or no control at the level of memory allocation and other hardware resources. We can plot different languages along this control that is safety axis, and it seems that when a language is safer, like Java compared to C++, it loses low-level control. The inverse is also true; a language that gives more control over resources like C++ provides much less safety.



Rust is made to overcome this dilemma by providing:

- High safety through its strong type system and smart compiler
- Deep but safe control over low-level resources (as much as C or C++), so it runs close to the hardware

Its main website, `http://www.rust-lang.org/en-US/`, contains links to installation instructions, docs and the Rust community.

Rust lets you specify exactly how your values are laid out in memory and how that memory is managed; that's why it works well at both ends of the control and safety line. This is the unique selling point of Rust, it breaks the safety-control dichotomy that, before Rust, existed among programming languages. With Rust they can be achieved together without losing performance.

Rust can accomplish both these goals without a garbage collector, in contrast to most modern languages like Java, C#, Python, Ruby, Go, and the like. In fact Rust doesn't even have a garbage collector yet (though an optional garbage collector is being designed).

Rust is a compiled language: the strict safety rules are enforced by the compiler, so they do not cause runtime overhead. As a consequence, Rust can work with a very small runtime, so it can be used for real-time or embedded projects and it can easily integrate with other languages or projects.

Rust is meant for developers and projects where performance and low-level optimizations are important, but also where a safe and stable execution environment is needed. The robustness of the language is specifically suited for projects where that is important, leading to less pressure in the maintenance cycle. Moreover, Rust adds a lot of high-level functional programming techniques to the language, so that it feels at the same time like a low-level and a high-level language.

The trifecta of Rust - safe, fast, and concurrent

Rust is not a revolutionary language with new cutting-edge features, but it incorporates a lot of proven techniques from older languages, while massively improving upon the design of C++ in matters of safe programming.

The Rust developers designed Rust to be a general purpose and multi-paradigm language; like C++, it is an imperative, structured and object-oriented language. Besides that, it inherits a lot from functional languages on the one hand, while also incorporating advanced techniques for concurrent programming on the other hand.

The typing of variables is static (because Rust is compiled) and strong. However, unlike in Java or C++, the developer is not forced to indicate types for everything; the Rust compiler is able to infer types in many cases.

C and C++ applications are known to be haunted by problems that often lead to program crashes or memory leaks, and which are notoriously difficult to debug and solve. Think about dangling pointers, buffer overflows, null pointers, segmentation faults, data races, and so on. The Rust compiler (called **rustc**) is very intelligent and can detect all these problems while compiling your code, thereby guaranteeing memory safety during execution. This is done by the compiler, retaining complete control over memory layout, but without needing the runtime burden of garbage collection (see `Chapter 6`, *Using Traits and OOP in Rust*). Of course, safety also implies much less possibility for security breaches.

Rust compiles to native code like Go and Julia but, in contrast to the other two, Rust needs no runtime with garbage collection. In this respect, it also differs from Java and the languages that run on the JVM, like Scala and Clojure. Most other popular modern languages, like .NET with C# and F#, JavaScript, Python, Ruby, Dart, and so on, all need a virtual machine and garbage collection for their execution.

Rust provides several mechanisms for concurrency and parallelism. The Standard Library gives a model that works with threads to perform work in parallel, where each thread maps to an operating system thread. They do not share heap memory, but communicate data through channels and data races are eliminated by the type system (see `Chapter 8`, *Organizing Code and Macros*). If needed in your project, several crates provide an actor-model approach with lightweight threads. These mechanisms make it easy for programmers to leverage the power of the many CPU cores available on current and future computing platforms.

The `rustc` compiler is completely self-hosted, which means it is written in Rust and can compile itself by using a previous version. It uses the LLVM compiler framework as its backend (for more info, see `http://en.wikipedia.org/wiki/LLVM`), and produces natively executable code that runs blazingly fast, because it compiles to the same low-level code as C++ (see some benchmarks at `http://benchmarksgame.alioth.debian.org/u64q/rust.php`).

Rust is designed to be as portable as C++ and to run on widely-used hardware and software platforms. At present, it runs on Linux, macOS X, Windows, FreeBSD, Android, and iOS. For a more complete overview of where Rust can run, see `https://forge.rust-lang.org/platform-support.html`.

Rust can call C code as simply and efficiently as calling C code from C itself, and, conversely C code can also call Rust code (see `Chapter 9`, *Concurrency - Coding for Multicore Execution*).

Rust developers are called **rustaceans**.

Other Rust characteristics that will be discussed, in more detail in the later chapters are as follows:

- Variables are immutable by default (see `Chapter 2`, *Using Variables and Types*)
- Enums (see `Chapter 4`, *Structuring Data and Matching Patterns*)
- Pattern matching (see also `Chapter 4`, *Structuring Data and Matching Patterns*)
- Generics (see `Chapter 5`, *Higher Order Functions and Error-Handling*)
- Higher-order functions and closures (see also `Chapter 5`, *Higher Order Functions and Error-Handling*)
- An interface system called **traits** (see `Chapter 6`, *Using Traits and OOP in Rust*)
- A hygienic macro system (see `Chapter 8`, *Organizing Code and Macros*)
- Zero-cost abstractions, which means that Rust has higher-language constructs, but these do not have an impact on performance

In conclusion, Rust gives you ultimate power over memory allocation, as well as removing many security and stability problems commonly associated with native languages.

Comparison with other languages

Dynamic languages such as Ruby or Python give you the initial speed of coding development, but the price is paid later in:

- Writing more tests
- Runtime crashes
- Production outages

The Rust compiler forces you to get a lot of things right from the beginning at compile time, which is the least expensive place to identify and fix bugs.

Rust's object orientation is not as explicit or evolved as common object-oriented languages such as Java, C# or Python, as it doesn't have classes. Compared with Go, Rust gives you more control over memory and resources and so it lets you code on a lower level. Go also works with a garbage collector; it has no generics and no mechanism to prevent data races between its `goroutines` used in concurrency. Julia is focused on numerical computing performance, works with a JIT compiler, and also doesn't give you that low-level control as Rust does.

The stability of Rust and its evolution

Rust started out with version 1.0.0, and, at the time of writing, the current version is 1.20.0. Version numbers follow the semantic versioning principle (see `http://semver.org/` for further information):

- **Patch release**: For bug fixes and other minor changes, increment the last number, for example 1.18.1
- **Minor release**: For new features which don't break existing features, increment the middle number, for example 1.19.0
- **Major release**: For changes which break backwards compatibility, increment the first number, for example 2.0.0

So, no breaking changes will occur during the current 1.n.m cycle versions, as this cycle is backward compatible; Rust projects which are developed in the older versions of this cycle will still compile in a more recent version. However, to be able to work with new features which are only contained in the more recent version, it is mandatory to compile your code to that specific version.

Rust has a very dynamic cycle of progression. Work is performed on three releases (called **channels** or builds simultaneously)--nightly, beta, and stable, and they follow a strict six-week release cycle like web browsers:

- The **stable channel** is the current stable release, which is advocated for Rust projects that are being used in production.
- The **beta channel** is where new features are deemed stable enough to be tested out in bigger, non-deployed projects.
- The **nightly channel** build contains the latest experimental features; it is produced from the master branch every 24 hours. You would use it only for experimentation.

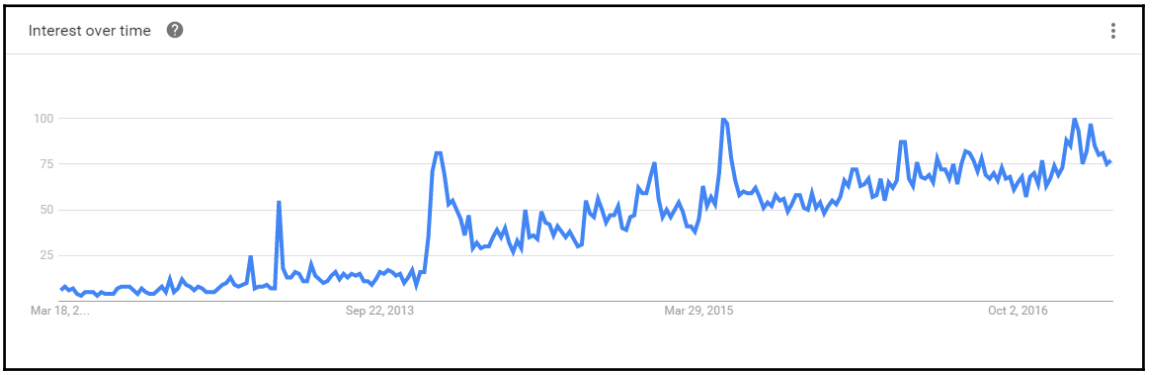
The beta and stable channel builds are only updated as new features are backported to their branch. With this arrangement, Rust allows users to access new features and bug fixes quickly.

Here is a concrete example: 1.18 was released on 18th June, 2017, the 1.19-beta was released at the same time, and the master development branch was advanced to 1.20. Six weeks later, on 30th July, Rust 1.19 will come out of beta and become a stable release, 1.20 will be promoted to 1.21-beta, and the master will become the eventual 1.21.

Some features in an experimental stage can only work when the code contains an attribute `#[feature]`. These may not be used on the stable release channel, only on a beta or nightly release; an example is the box syntax (see `chapter 2\code\references.rs`).

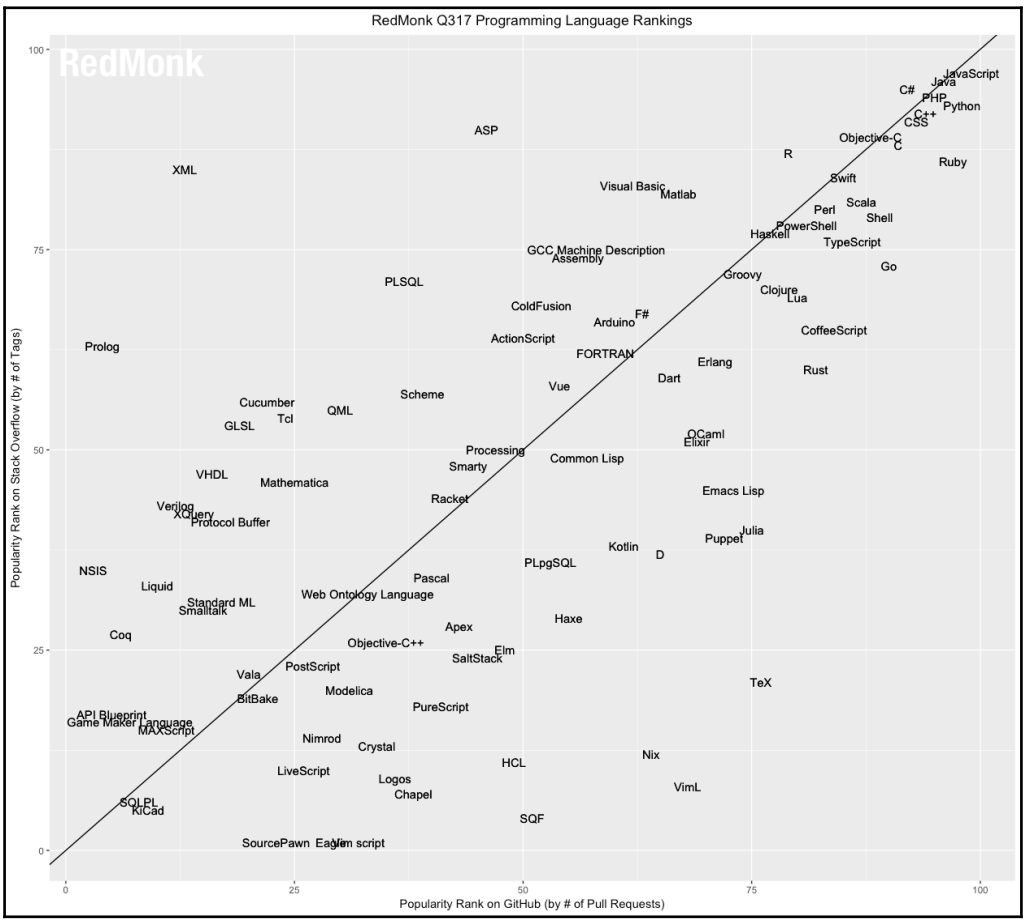
The success of Rust

Since its production release 1.0, Rust has enjoyed quite a steady uptake. This is manifest if you view a Google Trends survey:



In the well-known TIOBE Index (see `https://www.tiobe.com/tiobe-index//`), it reached 50th place in September 2015 and is now ranked in 37th position.

In the RedMonk ranking of programming languages (see `http://redmonk.com/sogrady/2017/06/08/language-rankings-6-17/`), it is ready to join the popularity of Lua, CoffeeScript, and Go.



Also, for two consecutive years, Rust was the most loved programing language on Stack Overflow (see `https://insights.stackoverflow.com/survey/2017#most-loved-dreaded-and-wanted`).

As a hallmark of its success, today, more than 50 companies are using Rust in production, see `https://www.rust-lang.org/en-US/friends.html`, amongst which are HoneyPot, Tilde, Chef, npm, Canonical, Coursera, and Dropbox.

Where to use Rust

It is clear from the previous sections that Rust can be used in projects that would normally use C or C++. Indeed, many regard Rust as a successor to, or a replacement for, C/C++. Although Rust is designed to be a systems language, due to its richness of constructs, it has a broad range of possible applications, making it an ideal candidate for applications that fall into one or all of the following categories:

- Client applications, like browsers
- Low-latency, high-performance systems, like device drivers, games and signal processing
- Highly distributed and concurrent systems, like server applications and microservices
- Real-time and critical systems, like operating systems or kernels
- Embedded systems (requiring a very minimal runtime footprint) or resource-constrained environments, like Raspberry Pi and Arduino, or robotics
- Tools or services that can't support the long warmup delays common in **just-in-time** (**JIT**) compiler systems and need instantaneous startup
- Web frameworks
- Large-scale, high-performance, resource intensive, and complex software systems

Rust is especially suitable when code quality is important, that is, for the following:

- Modestly-sized or larger development teams
- Code for long-running production use
- Code with a longer lifetime that requires regular maintenance and refactoring
- Code for which you would normally write a lot of unit tests to safeguard

Servo

Mozilla uses Rust as the language for writing Servo, its new web browser engine designed for parallelism and safety (`https://servo.org/`).

Due to Rust's compiler design, many kinds of browser security bugs are prevented automatically. In 2013, Samsung got involved, porting Servo to Android and ARM processors. Servo is itself an open source project with more than 750 contributors. It is under heavy development, and amongst other parts it already has its own CSS3 and HTML5 parser implemented in Rust. It passed the web compatibility browser test ACID2 in March 2014 (`http://en.wikipedia.org/wiki/Acid2/`).

Servo currently supports Linux, OS X, Windows, and Android. Parts of Servo are merged into Gecko (the engine on which Firefox is based), thus lending the Servo project's advancements to Firefox.

Installing Rust

You can install the Rust toolchain on every platform that Rust supports by using the `rustup` installer tool, which you can find at `http://www.rust-lang.org/install.html`.

On Windows, double-click on the `rustup-init.exe` file to install the Rust binaries and dependencies. Rust's installation directory (which by default is `C:\Users\username\.cargo\bin`) is automatically added to the search path for executables. Additionally you may need the C++ build tools for Visual Studio 2013 or later, which you can download from `http://landinghub.visualstudio.com/visual-cpp-build-tools`.

On Linux and OS X, run the following command in your shell:

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

This installs the Rust toolchain in `/home/username/.cargo/bin` by default.

Verify the correctness of the installation by showing Rust's version by typing `rustc -V` or `rustc - -version` in a console, which produces output like the following:

```
C:\Users\CVO>rustc -V
rustc 1.19.0 (0ade33941 2017-07-17)
```

Rust can be uninstalled by running the following command:

```
rustup self uninstall
```

The rustup tool enables you to easily switch between stable, beta, and nightly compilers and keep them updated. Moreover, it makes cross-compiling simpler with binary builds of the Standard Library for common platforms.

At `https://forge.rust-lang.org/platform-support.html` is a list of all the platforms on which Rust can run.

A bare metal stack called `zinc` for running Rust in embedded environments can be found at `http://zinc.rs/`; at this moment only the ARM architecture is supported.

The source code resides on GitHub (see `https://github.com/rust-lang/rust/`) and, if you want to build Rust from source, we refer you to `https://github.com/rust-lang/rust#building-from-source`.

rustc--the Rust compiler

The Rust installation directory containing `rustc` can be found on your machine in the following folder (unless you have chosen a different installation folder):

- On Windows at `C:\Users\username\.cargo\bin`
- On Linux or OS X in `/home/username/.cargo/bin`

`rustc` and the other binaries can be run from any command-line window. The `rustc` command has the following format:

```
rustc [options] input
```

The options are one-letter directives for the compiler after a dash, like `-g` or `-W`, or words prefixed by a double dash, like `- -test` or `- -version`. All options with some explanation are shown when invoking `rustc -h`. In the next section, we verify our installation by compiling and running our first Rust program.

To view a local copy of the Rust documentation as a website, type `rustup doc` into a terminal.

Our first program

Let's get started by showing a welcome message to the players of our game. Open your favorite text editor (like Notepad++ or gedit) for a new file and type in the following code:

```
// code in Chapter1\code\welcome.rs
fn main() {
    println!("Welcome to the Game!");
}
```

The steps to be performed are as follows:

1. Save the file as `welcome.rs`. The `.rs` extension is the standard extension of Rust code files. Source file names may not contain spaces; if they contain more than one word, you can use an underscore, _, as a separator, for example: `start_game.rs`.
2. Then compile it to native code on the command line with `rustc.welcome.rs`. This produces an executable program, `welcome.exe`, on Windows or `welcome` on Linux.
3. Run this program with `welcome` or `./welcome` to get the output:

```
Welcome to the Game!
```

The output executable gets its name from the source file. If you want to give the executable another name, like `start`, compile it with the option `-o output_name`, as shown below:

```
rustc welcome.rs –o start
```

The `rustc -O` produces native code optimized for execution speed (equivalent to `rustc -C opt-level=2`); the most optimized code is generated for `rustc -C opt-level=3`.

Compiling and running are separate consecutive steps, contrary to dynamic languages like Ruby or Python where these are performed in one step.

Let's explain the code a bit. If you have already worked in a C, or Java, or C# like environment, this code will seem quite familiar. As in most languages, execution of code starts in a `main()` function, which is mandatory in an executable program.

In a larger project with many source files, the file containing the `main()` function would be called `main.rs` by convention.

We see that `main()` is a function declaration because it is preceded by the keyword `fn`, short and elegant like most Rust keywords. The `()` after `main` denotes the parameter list, which is empty here. The function's code is placed in a code block, surrounded by curly braces `{ }`, where the opening brace is put by convention on the same line as the function declaration, but separated by one space. The closing brace appears after the code, in the column right beneath `fn`.

Our program has only one line, which is indented by four spaces to improve readability (Rust is not whitespace sensitive). This line prints the string `Welcome to the Game!`. Rust recognizes this as a string, because it is surrounded by double quotes `" "`. This string was given as argument to the `println!` macro (the `!` indicates it is a macro and not a function). The code line ends in a semicolon, `;`, as most, but not all, code lines in Rust do (see `Chapter 2`, *Using Variables and Types*).



Exercises:
Write, compile, and execute a Rust program, `name.rs`, that prints out your name.
What is the smallest possible program in Rust in terms of code size?

The `println!` macro has some nice formatting capabilities and at the same time checks when compiling whether the type of variables is correct for the applied formatting (see `Chapter 2`, *Using Variables and Types*).

Working with Cargo

Cargo is Rust's package and dependency manager, like Bundler, npm, pub, or pip for other languages. Although you can write Rust programs without it, Cargo is nearly indispensable for any larger project. It works the same whether you work on a Windows, Linux, or OS X system. The installation procedure from the previous section includes the Cargo tool, **cargo**, so Rust is shipped with the batteries included.

Cargo does the following things for you:

- It makes a tidy folder structure and some templates for your project, with the following command:

```
cargo new
```

- It compiles (builds) your code, using the following command:

```
cargo build
```

- It runs your project, using the following command:

```
cargo run
```

- If your project contains unit-tests, it can execute them for you, using the following command:

```
cargo test
```

- If your project depends on packages, it will download them and it will build these packages according to the needs of your code, using the following command:

```
cargo update
```

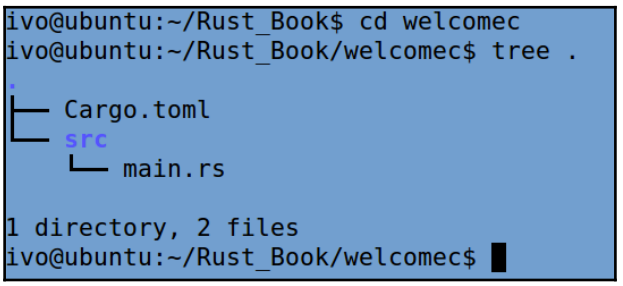
We'll introduce how to use Cargo now, and we'll come back to it later, but you can find more info at `http://doc.crates.io/guide.html`.

Let's remake our first project, `welcome`, using Cargo through the following steps:

1. Start a new project, `welcomec`, with the following command:

```
cargo new welcomec --bin
```

- The option `--bin` tells Cargo that we want to make an executable program (a binary). This outputs the message `Created binary (application) `welcomec` project` and creates the following directory structure:



- A folder with the same name as the project is created as a local Git project. In this folder, you can put all kinds of general info such as a License file, a README file, and so on. Also, a subfolder, `src`, is created, containing a template source file named `main.rs` (this contains the same code as our `welcome.rs`, but prints out the string `"Hello, world!"`).

- The file `Cargo.toml` (with a capital C) is the configuration file or manifest of your project; it contains all the metadata Cargo needs to compile your project. It follows the so called **TOML format** (for more details about this format, see `https://github.com/toml-lang/toml`), and contains the following text with information about the project:

```
[package]
name = "welcomec"
version = "0.1.0"
authors = ["Your name <you@example.com>"]
[dependencies]
```

- This file is editable and other sections can be added. For example, you can add a section to tell Cargo that we want a binary with name:

```
welcome:
[[bin]]
name = "welcome"
```

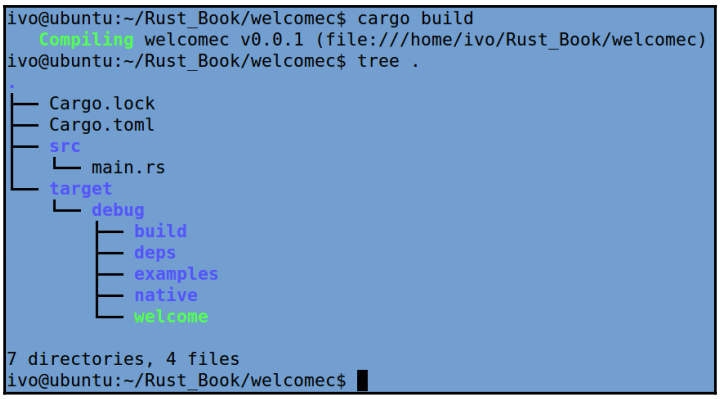
2. We build our project (no matter how many source files it contains) with the following command:

```
cargo build
```

- Which gives us the following output (on Linux):

```
Compiling welcomec v0.1.0 (file:///home/ivo/Rust_Book/welcomec)
Finished dev [unoptimized + debuginfo] target(s) in 0.66 secs
```

- Now, the following folder structure is produced:



- The `target/debug`directory contains the executable `welcome`.

3. To execute this program, give the following command:

```
cargo run
```

- Which produces as output:

```
Running `target/debug/welcome`
Hello, world!
```

Step 2 has also produced a file called `Cargo.lock`; this is used by Cargo to keep track of dependencies in your application. At this moment, it contains only the following:

```
[root]
name = "welcomec"
version = "0.1.0"
```

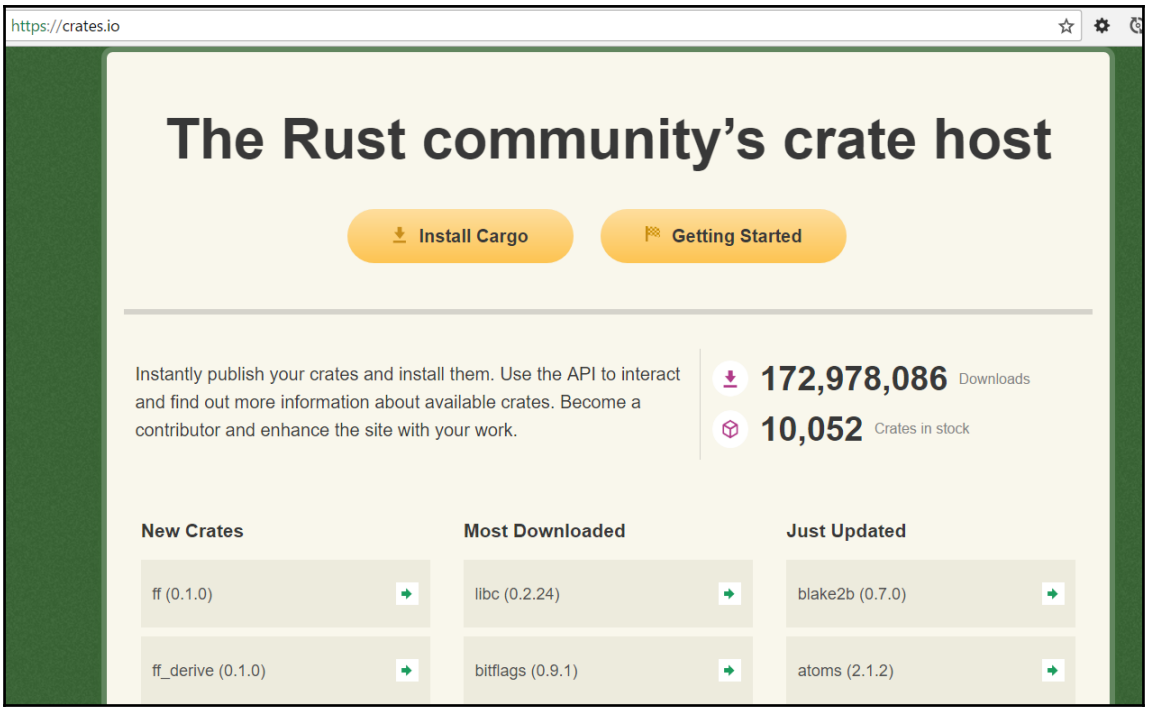
The same format is used to lock down the versions of libraries or packages your project depends on. If your project is built in the future, when updated versions of the libraries are available, Cargo will make sure that only the versions recorded in `Cargo.lock` are used, so that your project is not built with an incompatible version of a library. This ensures a repeatable build process.

The `cargo –list` gives you an overview of the commands you can use within this tool.



Exercise:
Make, build, and run a project, `name`, that prints out your name with Cargo.

The site `https://crates.io/` is the central repository for Rust packages, or crates as they are called, containing over 10000 crates at the end of June 2017. You can search for crates with specific terms, or browse them alphabetically or by number of downloads. The site looks like the following:



Developer tools

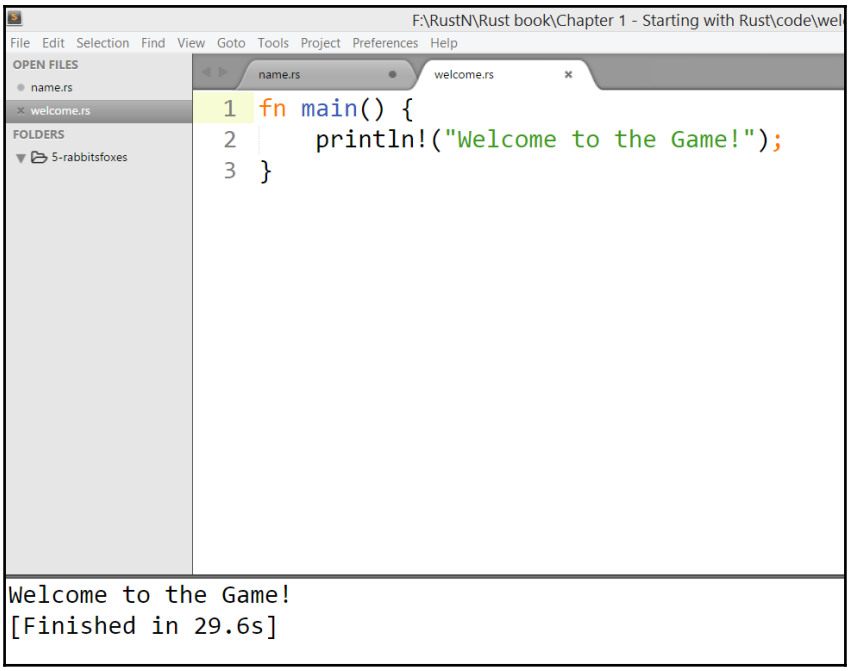
Because Rust is a systems programming language, the only thing you need is a good text editor (but not a word proecessor) for writing the source code, and everything else can be done using commands in a terminal session. However, some developers appreciate the functionalities offered by more fully-fledged text editors specific for programming or **Integrated Development Environments** (**IDEs**). Rust has a lot of possibilities in this regard. Most Rust developers work with Vim or Emacs, but Rust plugins exist for a host of text editors, like Atom, Brackets, BBEdit, Emacs, Geany, gedit, Kate, TextMate, Textadept, Vim, NEdit, Notepad++, Sublime Text, and Visual Studio Code. Also, some IDEs, such as Eclipse (`RustDT`), Netbeans (`rust-netbeans`), IntelliJ, and Visual Studio, provide plugins for Rust; see the updated overview at `https://github.com/rust-unofficial/awesome-rust#ides`.

These come with a varying range of features, such as syntax highlighting, code formatting, code completion, linting, debugging, Cargo project support, and so on.

Using Sublime Text

The plugins for the popular Sublime Text editor (`http://www.sublimetext.com/3`) are particularly pleasant to work with and don't get in your way. After having installed Sublime Text (you might want to get a registered version if you start using it regularly), also install the Package Control package (for instructions on how to do that visit `https://packagecontrol.io/installation`).

Then, to install the Sublime Text Rust plugin, open the command palette in Sublime Text (*Ctrl+Shift+P* on Windows or cmd+*Shift+P* on OS X) and select **Package Control** | **Install Package** and then select **RustEnhanced** from the list.



Sublime Text is a very complete text editor, including color schemes. The Rust plugin provides syntax highlighting and auto-completion; type one or more letters, choose an option from the list that appears with an arrow key and then press *Tab* to insert the code snippet, or simply select a list option through a mouse click.

When working with Rust code, select **Tools** | **Build System** and **RustEnhanced**.

Then, you can run and compile a source file with *Ctrl+B*. Warnings or errors appear in the lower panel; if everything is OK the output of the program appears together with a message like the following:

```
[Finished in 0.6s]
```

If you want to do the two steps separately, do *Ctrl+Shift+B*. A pop-up menu appears, click on **RustEnhanced** if you only want to compile, click on **RustEnhanced | Run** if you want to execute the program. A `SublimeLinter` plugin exists that provides an interface to `rustc`, called `SublimeLinter-contrib-rustc`. It does additional checks on your code for stylistic or programming errors. You can install it as indicated above through Package Control and then use it from the menu **Tools | SublimeLinter** (for more details consult `https://github.com/oschwald/SublimeLinter-contrib-rustc`).

There are also plugins for IDEs like:

- For Eclipse, called **RustDT**: `https://github.com/RustDT/RustDT`
- For IntelliJ called **intellij-rust**: `https://github.com/intellij-rust/intellij-rust`
- For NetBeans called **rust-netbeans**: `https://github.com/drrb/rust-netbeans`
- For Visual Studio called **VisualRust**: `https://github.com/PistonDevelopers/VisualRust`

You can test out Rust code even without local installation with the Rust Playground at `http://play.rust-lang.org/`, which allows you to edit or paste your code and evaluate it.

The interactive shell rusti or **Read-Evaluate-Print-Loop** (**REPL**) is in development for Rust, which is common for dynamic languages, but remarkable for a statically compiled language. You can find it at `https://github.com/murarth/rusti`.

An online environment that combines both is `repl.it`. Refer the following link for more details `https://repl.it/languages/rust`.

The Standard Library

Rust's Standard Library, `stdlib`, contains all primitive types, basic modules, and macros. In fact, nearly all of this book talks about it, with `Chapter 11`, *Exploring the Standard Library* filling in some gaps.

It is the well-tested and minimal code that ensures portability to a wide diversity of platforms and on which is built the rest of the ecosystem.

Having installed Rust also means that you have a binary version of the Standard Library on your system. When you compile source code or do a `cargo build`, this `stdlib` is included; this explains why the executable file size is not that small (for example 129 KB for `welcomec.exe` on Windows).

Compile with `rustc -C prefer-dynamic welcome.rs` to get a small executable, like 10 KB for `welcome`.

Summary

In this chapter, we got an overview of Rust's characteristics, where Rust can be applied and compared it to other languages. We made our first program, demonstrated how to build a project with Cargo, and gave you choices on how to make a more complete development environment.

In the following chapter, we will look at variables and types, and explore the important concept of mutability.

2
Using Variables and Types

In this chapter, we look at the basic building blocks of a Rust program, like variables and types. We discuss variables of primitive types, whether their type has to be declared or not, and the scope of variables. Immutability, one of the cornerstones of Rust's safety strategy, is also discussed and illustrated.

We will cover the following topics:

- Comments
- Global constants
- Values and primitive types
- Binding variables to values
- Scope of a variable and shadowing
- Type checking and conversions
- Expressions
- The stack and the heap

Our code examples will center on building a text-based game called **Monster Attack**.

Comments

Ideally, a program should be self-documenting by using descriptive variable names and easy-to-read code, but there are always cases where additional comments about a program's structure or algorithms are needed. Rust follows the C convention and has:

- `//` line comments; everything on the line after `//` is commentary and not compiled
- `/* */`block or multi-line comments; everything between the start `/*`and the end `*/` is not compiled

However, the preferred Rust style is to use only the `//` comment, also for multiple lines, as shown in the following code:

```
// see Chapter 2/code/comments.rs
fn main() {
  // Here starts the execution of the Game.
  // We begin with printing a welcome message:
  println!("Welcome to the Game!");
}
```

Use the `/* */` comments only to comment out code.

Rust also has a doc comment with `///`, useful in larger projects that require an official documentation for customers and developers. Such comments have to appear before an item (like a function) on a separate line to document that item. In these comments, you can use Markdown formatting syntax (see `https://en.wikipedia.org/wiki/Markdown`).

Here is a doc comment:

```
/// Start of the Game
fn main() {
}
```

We'll see more relevant uses of `///` in later code snippets. The rustdoc tool can compile these comments into project documentation.

Global constants

Often, an application needs a few values that are in fact constants, meaning that they do not change in the course of the program. In our game, for example, the game name `Monster Attack` could be a constant, as could the maximum health amount, which is the number `100`. We must be able to use them in the `main()` function or any other function in our program, so they are placed at the top of the code file. They live in the global scope of the program. Such constants are declared with the keyword `static`, as follows:

```
// see Chapter 2/code/constants1.rs
static MAX_HEALTH: i32 = 100;
static GAME_NAME: &str = "Monster Attack";

fn main() {
}
```

Names of constants must be in uppercase, underscores can be used to separate word parts. Their type must also be indicated; the variable `MAX_HEALTH` is a 32-bit integer (`i32`) and the variable `GAME_NAME` is a string (`str`) type. As we will discuss further, the declaration of types for variables is done in exactly the same way, although it is often optional when the compiler can infer the type from the code context.

Remember that Rust is a low-level language, so many things must be specified in detail. The `&` is a reference to something (it contains its memory address), here of the string.

The compiler gives us a warning, which looks like this:

```
warning: static item is never used: `MAX_HEALTH`, #[warn(dead_code)] on by
default
```

This warning does not prevent the compilation, so in this stage, we can compile to an executable `constants1.exe`. But the compiler is right; these objects are never used in the program's code (they are called dead code), so, in a complete program, either use them or throw them out.



It takes a while before the aspiring Rust developer starts to regard the Rust compiler as his friend, and not as an annoying machine spitting out errors and warnings. As long as you see this message at the end of the compiler output,`error: aborting due to previous errors`, no (new) compiled executable is made. But remember, correcting the errors eliminates runtime problems, so this can save you a lot of time otherwise wasted tracking nasty bugs. Often, the error messages are accompanied by helpful notes on how to eliminate the error. Even the warnings can point you to flaws in your code. Rust also warns us when something is declared but not used in the code that follows, like unused variables, functions, imported modules, and so on. It even warns us if we make a variable mutable (which means it's value can be changed) when it should not be, or when code doesn't get executed! The compiler does such a good job that when you reach the stage that all errors and warnings are eliminated, your program most likely will run correctly!

Besides static values, we can also use simple constant values whose value never changes. Constants always have to be typed, as shown here:

```
const MYPI: f32 = 3.14;
```

The compiler automatically substitutes the value of the constant everywhere in the code.

Printing with string interpolation

An obvious way to use variables is to print out their value, as is done here:

```
// see Chapter 2/code/constants2.rs
static MAX_HEALTH: i32 = 100;
static GAME_NAME: &str = "Monster Attack";
const MYPI: f32 = 3.14;

fn main() {
  println!("The Game you are playing is called {}.", GAME_NAME);
  println!("You start with {} health points.", MAX_HEALTH);
}
```

This gives an output which looks like this:

```
The Game you are playing is called Monster Attack.
You start with 100 health points.
```

The first argument of the `println!` macro is a literal format string containing a placeholder `{}`. The value of the constant or variable after the comma is converted to a string and comes in its place. There can be more than one placeholder and they can be numbered in order, so that they can be used repeatedly, as in the following code:

```
println!("In the Game {0} you start with {1} % health, yes you read it
correctly: {1} points!", GAME_NAME, MAX_HEALTH);
```

This produces the following output:

```
In the Game Monster Attack you start with 100 % health, yes you read it
correctly: 100 points!
```

The placeholder can also contain one or more named arguments, like this:

```
println!("You have {points} % health", points = 70);
```

This produces the following output:

```
You have 70 % health
```

Special ways of formatting can be indicated inside the `{}` after a colon (`:`), optionally prefixed by a position, like this:

```
println!("MAX_HEALTH is {:x} in hexadecimal", MAX_HEALTH); //
```

This gives an output like this: `64`

```
println!("MAX_HEALTH is {:b} in binary", MAX_HEALTH);   //
```

This gives an output like this: `1100100`

```
println!( "Two written in binary is {0:b}", 2);   //
```

This gives an output like this: `10`

```
println!("pi is {:e} in floating point notation", PI);  //
```

This gives an output like this: `3.14e0`.

The following formatting possibilities exist:

- **o**: For octal
- **x**: For lower hexadecimal
- **X**: For upper hexadecimal
- **p**: For a pointer
- **b**: For binary

- **e**: For lower exponential notation
- **E**: For upper exponential notation
- **?**: For debugging purposes

The `format!` macro has the same parameters and works the same way as the `println!` macro, but it returns a string instead of printing out.



Consult `http://doc.rust-lang.org/std/fmt/` for an overview of all possibilities.

Values and primitive types

Constants that have been initialized have a value. Values exist in different types: `70` is an integer, `3.14` is a `float`, and `Z` and `θ` are the type of a character. Characters are Unicode values that take four bytes of memory each. `Godzilla` is a string of type `&str` (which is Unicode UTF8 by default), `true` and `false` are the type of Boolean values. Integers can be written in different formats:

- Hexadecimal format with `0x`, like `0x46` for `70`.
- Octal format with `0o`, like `0o106` for `70`.
- Binary format with `0b`, like `0b1000110`.

- Underscores can be used for readability, as in `1_000_000`. Sometimes the compiler will urge you to indicate more explicitly the type of number with a suffix, for example (the number after `u` or `i` is the number of memory bits used, namely: 8, 16, 32, or 64).
- The `10usize` denotes an unsigned integer of machine word size (`usize`), which can be any of the following types: `u8`, `u16`, `u32`, `u64`.
- The `10isize` denotes a signed integer of machine word size (`isize`), which can be any of the following types: `i8`, `i16`, `i32`, `i64`
- In the cases above on a 64-bit operating system `usize` is in fact `u64`, and `isize` is equivalent to `i64`.
- The `3.14f32` denotes a 32-bit floating-point number.

- The `3.14f64` denotes a 64-bit floating-point number.
- The numeric types `i32` and `f64` are the defaults if no suffix is given, but in that case to differentiate between them you must end an `f64` value with `.0`, like:

```
let e = 7.0;
```

In general, indicating a specific type is recommended.

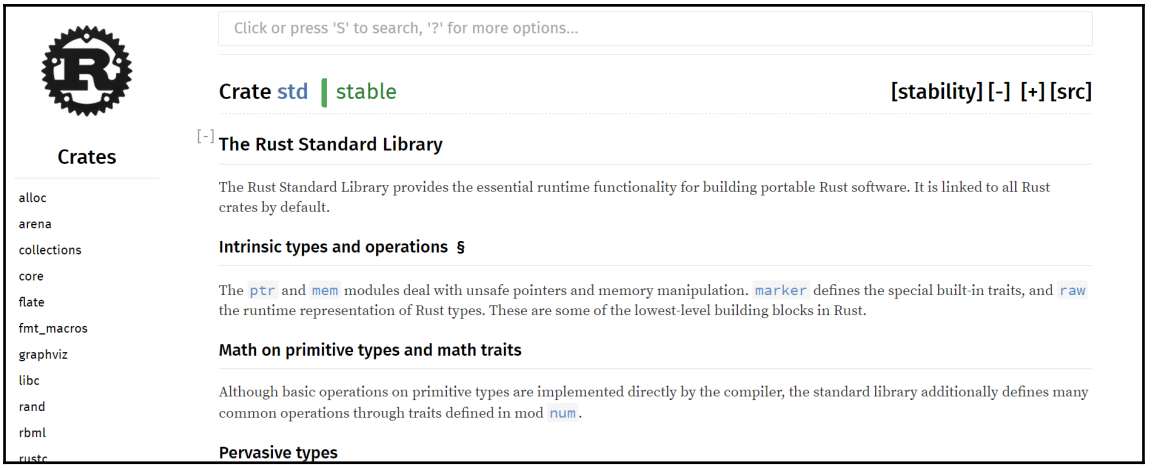
Rust is like any other C-like language when it comes to the different operators and their precedence. However, notice that Rust does not have increment (++) or decrement (--) operators. To compare two values for equality use == , and `!=` to test if they are different.

There is even the empty value `()` of zero size, which is the only value of the so called unit type `()`. It is used to indicate the return value when an expression or a function returns nothing (no value), as is the case for a function that only prints to the console. `()` is not the equivalent of a null value in other languages; `()` is no value, whereas null is a value.

Consulting Rust documentation

The quickest way to find more detailed information about a Rust topic is to browse to the documentation screen of the Standard Library, `http://doc.rust-lang.org/std/`.

On the left, you can find a listing of all crates available, which you can browse for more details. But most useful is the search box at the top: type in a few letters or a word to get a number of useful references.





Exercises:
Try to change the value of a constant. Of course this is not allowed, what error do you get? (For an example see `Chapter2/exercises/change_constant.rs`).
Look up the `println!` macro in the documentation.
Read the `fmt` specification and write a program that will print value `3.2f32` as `+003.20` (see `Chapter2/exercises/formatting.rs`).

Binding variables to values

Storing all values in constants is not an option. It is not good because constants live as long as the program and moreover can't change, and often we want to change values. In Rust, we can bind a value to a variable by using a `let` binding.

```
// see Chapter 2/code/bindings.rs
fn main() {
  let energy = 5; // value 5 is bound to variable energy
}
```

Unlike in many other languages, such as Python or Go, the semicolon,`;`, is needed here to end the statement. Otherwise, the compiler throws an error, as follows:

```
error: expected one of `.`, `;`, or an operator, found `}`
```

We also want to create bindings only when they are used in the rest of the program, but don't worry, the Rust compiler warns us about that. The warning looks like the following:

```
values.rs:2:6: 2:7 warning: unused variable: `energy`,
#[warn(unused_variables)] on by default
```

For prototyping purposes, you can suppress that warning by prefixing the variable name with an _, like in `let _ energy = 5;` in general _ is used for variables we don't need.

Notice that in the declaration above we did not need to indicate the type. Rust infers the type of the `energy` variable to be an integer; the `let` binding triggers that. If the type is not obvious, the compiler searches in the code context where the variable gets a value or how it is used.

But giving type hints like `let energy = 5u16;` is also OK; that way, you help the compiler a bit by indicating the type of `energy`, in this case a two-byte unsigned integer.

We can use the variable `energy` by using it in an expression, for example assigning it to another variable, or printing it:

```
let copy_energy = energy;
println!("Your energy is {}", energy););
```

Here are some other declarations:

```
let level_title = "Level 1";
let dead = false;
let magic_number = 3.14f32;
let empty = ();  // the value of the unit type ()
```

The value of the `magic_number` variable could also be written as `3.14_f32`; the _ separates the digits from the type to improve readability.

Declarations can replace previous declarations of the same variable. Consider a statement like the following:

```
let energy = "Abundant";
```

It would now bind the variable `energy` to the value `Abundant` of type string. The old declaration can no longer be used and its memory is freed.

Mutable and immutable variables

Suppose we get a boost from swallowing a health pack and our energy rises to a value of `25`. However, if we assign the value to the variable `energy` as follows:

```
energy = 25;
```

We get an error, as follows:

```
error: re-assignment of immutable variable `energy`.
```

What is wrong here?

Well, Rust applies programmer's wisdom here: a lot of bugs come from inadvertent or wrong changes of variables, so don't let code change a value unless you have deliberately allowed it!



Variables are, by default,immutable, in Rust, which is very similar to what functional languages do (in pure functional languages, mutability is not even allowed).

If you want a mutable variable, because its value can change during code execution, you have to indicate that explicitly with the `mut` variable, for example:

```
let mut fuel = 34;
fuel = 60;
```

Simply declaring a variable as `let n;` is also not enough. We get the following error:

```
error: type annotations needed, consider giving `energy2` a type, cannot
infer type for `_`
```

Indeed, the compiler needs a value to infer its type.

We can give the compiler this information by assigning a value to the variable `n`, like `n = -2;` but as the message says, we could also indicate its type as follows:

```
let n: i32;
```

Or:

```
let n: i32 = -2; // n is a binding of type i32 and value -2
```

The type (here `i32`) follows the variable name after a colon `:` (as we already showed for global constants), optionally followed by an initialization. In general the type is indicated like this-`n: T` where `n` is a variable and `T` a type, and it reads, variable `n` is of type `T`. So, this is the inverse of what is done in C or C++, Java or C#, where one would write `Tn`.

For the primitive types, this can also be done simply with a suffix, as follows:

```
let x = 42u8;
let magic_number = 3.14f64;
```

Trying to use an uninitialized variable results in the following error:

```
error: use of possibly uninitialized variable
```

Local variables have to be initialized before use in order to prevent undefined behavior. When the compiler does not recognize a name (for example, a function name) in your code, you will get the following error:

```
error: not found in this scope error
```

It is probably just a typo, but it is caught early on at compilation, and not at runtime!

Scope of a variable and shadowing

All variables defined in the program `bindings.rs` have local scope delimited by the `{ }` of the function which happens to be the `main()` function here, but this applies to any function. After the ending, `},` they go out of scope and their memory allocation is freed.

We can even make a more limited scope inside a function by defining a code block as all code contained within a pair of curly braces `{ }`, as in the following snippet:

```
// see Chapter 2/code/scope.rs
fn main() {
  let outer = 42;
   { // start of code block
        let inner = 3.14;
        println!("block variable: {}", inner);
        let outer = 99; // shadows the first outer variable
        println!("block variable outer: {}", outer);
    } // end of code block
    println!("outer variable: {}", outer);
}
```

The preceding code gives the following output:

```
block variable: 3.14
block variable outer: 99
outer variable: 42
```

A variable defined in the block (like `inner`) is only known inside that block. A variable in the block can also have the same name as a variable in an enclosing scope (like `outer`), which is replaced (shadowed) by the block variable until the block ends. What do you expect when you try to print out `inner` after the block? Try it out.

Why would you want to use a code block? In the section *Expressions*, we will see that a code block can return a value that can be bound to a variable with the `let` binding. A code block can also be empty as `{ }`.

Type checking and conversions

Rust has to know the type of variables, because that way it can check (at compile time) that they are only used in ways their type permits. That way programs are type safe and a whole range of bugs are avoided.

This also means that we cannot change the type of a variable during its lifetime, because of static typing, for example, the variable `score` in the following snippet cannot change from an integer to a string:

```
// see Chapter 2/code/type_errors.rs
// warning: this code does not work!
fn main() {
let score: i32 = 100;
score = "YOU WON!"
}
```

With this, we get the compiler error, as follows:

```
error: mismatched types: expected i32, found reference
```

However, I am allowed to write this as:

```
let score = "YOU WON!";
```

Rust lets me redefine variables; each `let` binding creates a new variable score that hides the previous one, which is freed from memory. This is actually quite useful because variables are immutable by default.

Adding strings with + (like the players in the following code) is not defined in Rust:

```
let player1 = "Rob";
let player2 = "Jane";
let player3 = player1 + player2;
```

With this we get an error stating:

```
error: binary operation `+` cannot be applied to type `&str`
```

In Rust you can use the `to_string()` method to convert the value to a String type like this:

```
let player3 = player1.to_string() + player2;
```

Alternatively, you could use the `format!` macro:

```
let player3 = format!("{}{}", player1, player2);
```

In both cases, the variable `player3` has the value `RobJane`.

Let's find out what happens when you assign a value from a variable of a certain type to another variable of a different type:

```
// see Chapter 2/code/type_conversions.rs
fn main() {
  let points = 10i32;
  let mut saved_points: u32 = 0;
   saved_points = points; // error !
}
```

This is again not allowed, therefore, we get the same error, as follows:

```
error: mismatched types: expected u32, found i32
```

To enable maximal type checking Rust does not permit automatic (or implicit) conversions of one type to another like C++ does, thus avoiding a lot of hard-to-find bugs. For example, the numbers after the decimal point are lost when converting an `f32` value to an `i32` value; this could lead to errors when done automatically.

We can however do an explicit conversion (also called a casting) with the keyword as follows:

```
saved_points = points as u32;
```

When the variable `points` contains a negative value, the sign would be lost after conversion. Similarly, when casting from a wider value like a float to an integer, the decimal part is truncated. See the following example:

```
  let f2 = 3.14;
  saved_points = f2 as u32; // truncation to value 3 occurs here
```

Also, the value must be convertible to the new type, a string cannot be converted to an integer, for example:

```
  let mag = "Gandalf";
   saved_points = mag as u32; //
```

This gives an error, as follows:

```
error: non-scalar cast:`&str`as`u32`
```

Aliasing

It can be useful sometimes to give a new, more descriptive, or shorter name to an existing type. This is done with the keyword `type`, as in the following example where we needed a specific (but size-limited) variable for `MagicPower`:

```
// see Chapter 2/code/alias.rs
type MagicPower = u16;

fn main() {
  let mut run: MagicPower= 7800;
}
```

A `type` name starts with a capital letter, as does each word part of the name.

What happens when we change the value `7800` to `78000`? The compiler detects this with the following warning:

```
warning: literal out of range for u16
```

Expressions

Rust is an expression-oriented language, which means that most pieces of code are in fact expressions, that is, they compute a value and return that value. However, expressions on themselves do not form meaningful code; they must be used in statements.

`let` bindings like the following are declaration statements; they are not expressions:

```
// see Chapter 2/code/expressions.rs
let a = 2;    // a binds to 2
let b = 5;    // b binds to 5
let n = a + b;   // n binds to 7
```

But, here, `a + b` is an expression and, if we omit the semicolon at the end, the resulting value (here the value `7`) is returned. This is often used when a function needs to return its value (see examples in the next chapter). Ending an expression with a semicolon like `a + b;` suppresses this behaviour, thus throwing away the return value and making it an expression statement returning the unit value `()`.

Code is usually a sequence of statements, one on each code line and Rust has to know when a statement ends, that's why nearly every Rust code line ends with a semicolon.

What do you think the assignment `m = 42;` is? It is not a binding, because there is no `let` binding (that should have happened on a previous code line). It is an expression that returns the unit value `()`.

A compound binding like `let p = q = 3;` is not allowed in Rust, it returns the following error:

```
error: unresolved name q
```

However, you can chain `let` bindings like this:

```
let mut n = 0;
let mut m = 1;
let t = m; m = n; n  = t;
println!("{} {} {}", n, m, t); //
```

This gives an output as: `1 0 1`



Exercise:
Print out the values of `a`, `b`, and `n` after this code snippet. Explain the value of `a`(For an example code, see`compound_let.rs`):

```
let mut a = 5;
let mut b = 6;
let n = 7;
let a = b = n;
```

A code block is also an expression, which returns the value of its last expression if we omit the semicolon. For example, in the following code snippet, `n1` gets the value `7`, but `n2` gets no value (or, rather, the unit value `()`), because the return value of the second block was suppressed:

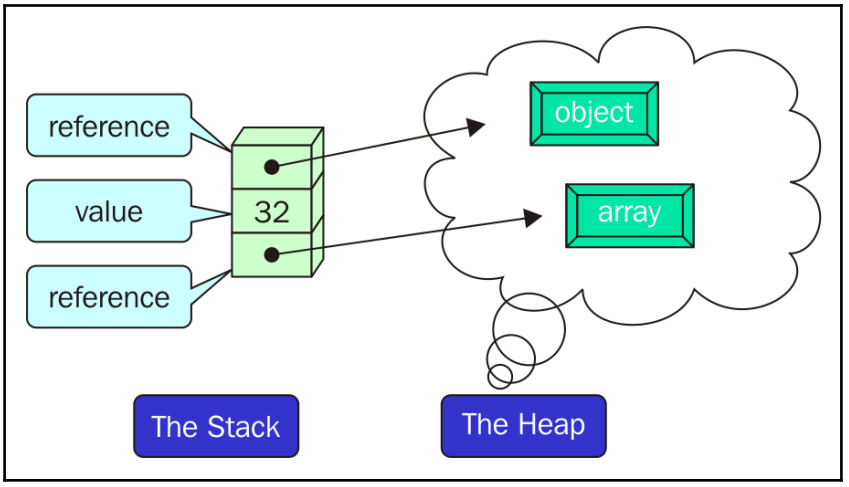
```
let n1 = {
let a = 2;
let b = 5;
a + b   // <-- no semicolon!
};
  println!("n1 is: {}", n1);  // prints: n1 is 7
  let n2 = {
    let a = 2;
    let b = 5;
    a + b;
  };
  println!("n2 is: {:?}", n2);  // prints: n2 is ()
```

The variables `a` and `b` are here declared in a code block and live only as long as the block itself, they are local to the block. Note that the semicolon after the closing brace of the block, `};`, is needed. To print out the unit value `()`, we need `{:?}` as format specifier.

The stack and the heap

Because memory allocation is very important in Rust, we must have a good mental picture of what is going on. A program's memory is divided into the stack and heap memory parts; to get more background on these concepts go to`https://stackoverflow.com/questions/79923/what-and-where-are-the-stack-and-heap`.

Primitive values as numbers (like **32** in the figure), characters, or `true` or `false` values are stored on the stack, but the value of more complex objects that could grow in size are stored in heap memory. Heap values are referenced by a variable on the stack, which contains the memory address of the object on the heap.



While the stack has a limited size, the size of the heap can grow as much as the space needed.

Suppose we run the following program and try to visualize the program's memory:

```
// see Chapter 2/code/references.rs
let health = 32;
let mut game = "Space Invaders";
```

Values are stored in memory and so have memory addresses. The variable `health` contains an integer value `32` which is stored in the stack on location `0x23fba4`, while the variable `game` contains a string, which is stored in the heap starting on location `0x23fb90` (these were the addresses when I executed the program, they will be different when you run the program).

The variables to which the values are bound are pointers or references to these values, they point to them. The variable `game` is a reference to `"Space Invaders"`. The address of a value is given by the `&` operator. So, the `&health` pointer is the address where value `32` is stored, and the `&game` pointer is the address where value `Space Invaders` is stored.

We can print these addresses by using the format string `{:p}` for pointers, like this:

```
println!("address of health-value: {:p}", &health);
// prints 0x23fba4
  println!("address of game-value: {:p}", &game); // prints 0x23fb90
println!("game-value: {}", game); // prints "Space Invaders"
```

Now, we have the following situation in memory (memory addresses will be different at each execution):

	memory location	**name**	**value**
H	0x23fb90		"Space Invaders"
E			
A			
P			
S			
T	0x23fba4		32
A	0x08	health	0x23fba4
C	0x04	game	0x23fb90
K		game2	0x23fb90

We can make an alias, which is another reference that points to the same place in memory, like this:

```
let game2 = &game;
println!("{:p}", game2); // prints 0x23fb90
```

To get the value being referred to rather than the reference `game2` itself, dereference it with the asterisk `*` operator, like this:

```
println!("{}", *game2); // prints "Space Invaders"
```

The `println!` macro is clever, so `println!("{}", game2);` will also print out the same value, as the following statement does:

```
println!("game: {}", &game);
```

The story above is a bit simplified, because Rust will allocate values that do not change in size as much as possible on the stack, but it is meant to give you a better idea of what a reference to a value means.

We know already that a `let` binding is immutable, so the value cannot be changed as `health = 33;`.

This gives an error, as follows:

```
error: re-assignment of immutable variable `health`
```

If the variable `y` is declared with `let y = &health;` then the reference `*y` is the value `32`. Reference variables can also be given a type like `let x: &i64;` and such references can be passed around in code. After this `let` binding `x` does not really point yet to a value, it does not contain a memory address. In Rust, there is no way to create a null pointer as you can in other languages, trying to assign a `nil` or `null` or even unit value `()` to `x` results in an error. This alone saves Rust programmers from countless bugs. Furthermore, trying to use the variable `x` in an expression, for example, the statement:

```
println!("{:?}", x);
```

This results with an error, as follows:

```
error: use of possibly uninitialized variable: `x`
```

A mutable reference (indicated as the `&mut` pointer) to an immutable variable is forbidden, otherwise the immutable variable could be changed through its mutable reference:

```
let tricks = 10;
let reftricks = &mut tricks;
```

This gives an error, as follows:

```
cannot borrow immutable local variable `tricks` as mutable
```

A reference to a mutable variable `score` can either be immutable or mutable, like the `score2` and `score3` variables respectively in the example below:

```
let mut score = 0;
let score2 = &score;
```

But you cannot change the value of `score` through an immutable reference to the `score2` variable, as this gives an error, as follows:

```
*score2 = 5;
This gives the an error: cannot assign to immutable borrowed content
*score2
```

The value of the variable `score` can only be changed through a mutable reference like `score3`:

```
let mut score = 0;
let score3 = &mut score;
*score3 = 5;
```

For reasons we will see later, you can only make one mutable reference to a mutable variable:

```
let score4 = &mut score;
```

If you do this, an error is thrown as follows:

```
error: cannot borrow `score` as mutable more than once at a time
```

Here, we touch at the heart of Rust's memory safety system, borrowing a variable is one of its key concepts. We will explore this in more detail in the `Chapter 7`, *Ensuring Memory Safety and Pointers*.

The heap is a much larger memory part than the stack, so it is important that memory locations are freed as soon as they are no longer needed. Every variable in Rust has a certain lifetime, which says how long the variable lives in the program's memory. The Rust compiler sees when a variable's lifetime has come to an end (or in other words, the variable goes out of scope), and inserts code at compilation time to free its memory when executing that code. This behavior is unique to Rust and is not done in other commonly used languages.

Stack values can be boxed, that is, allocated in the heap, by creating a `Box` around them, as is the case for the value of `x` in:

```
let x = Box::new(5i32);
```

The object `Box` references a value on the heap. We'll also look at it more closely in the section *Boxes* in `Chapter 7`, *Ensuring Memory Safety and Pointers*.

Summary

In this chapter, we learned how to work with variables in Rust, getting acquainted with many of the common compiler error messages. We explored types and the default immutability of variables as cornerstones of Rust's safety behavior.

In the following chapter, we will start writing some useful code using program logic and functions.

3
Using Functions and Control Structures

This chapter concentrates on how we can control the execution flow of our code and how to modularize our code through functions. We also learn how to document and test our code.

We will cover the following topics:

- Branching on a condition
- Looping
- Functions
- Attributes
- Testing

Branching on a condition

Branching on a condition is done with a common `if`, `if else`, or `if else if else` construct, as in this example:

```
// from Chapter 3/code/ifelse.rs
fn main() {
  let dead = false;
  let health = 48;
  if dead {
    println!("Game over!");
       return;
  }
  if dead {
    println!("Game over!");
```

```
      return;
    } else {
      println!("You still have a chance to win!");
    }
    if health >= 50 {
        println!("Continue to fight!");
    } else if health >= 20  {
        println!("Stop the battle and gain strength!");
    } else {
        println!("Hide and try to recover!");
    }
}
```

This gives the following output:

```
You still have a chance to win!
Stop the battle and gain strength!
```

The condition after the `if` statement has to be a Boolean. However, unlike in C, the condition must not be enclosed in parentheses. Code blocks surrounded by `{ }` (curly braces) are needed after the `if`, `else`, or `else if` statement. The first example also shows that we can get out of a function with the `return` value.

Also the `if else` condition is an expression that returns a value. This value can be used as a function call parameter in a `print!` statement, or it can be assigned in a `let` binding, like this:

```
let active = if health >= 50 {
          true
        }else{
          false
        };
println!("Am I active? {}", active);
```

This prints the following output:

```
Am I active? false
```

The code blocks could contain many lines, but be careful: when returning a value, you must omit the `;` (semi-colon) after the last expression in the `if` or `else` block (see section *Expressions* in `Chapter 2`, *Using Variables and Types*). Moreover, all branches always must return a value of the same type.

This also alleviates the need for a ternary operator (`?:`), like in C++; simply use `if,` as follows:

```
let adult = true;
let age = if adult { "+18" } else { "-18" };
println!("Age is {}", age);  //
```

This gives the following output:

```
Age is +18
```

Exercise:



1. See code in `Chapter 3/exercises/iftest.rs`.
2. Try adding a `;` (semi-colon) after the `+18` and `-18` values, like this `{"+18";}`, what value will be printed for the variable `age`? What happens if you type annotate the variable `age` as `&str`?
3. See if you can omit the `{ }` (curly braces) if there is only one statement in the block.
4. Also, verify if this code is OK:
   ```
   let health = -3;
   let result = if health <=0 { "Game over man!" };
   ```

How would you correct this statement if necessary?--by using pattern matching, which we will examine in the next chapter, also branches code, but does so based on the value of a variable.

Looping

For repeating pieces of code, Rust has the common `while` loop, again without parentheses around the condition:

```
// from Chapter 3/code/loops.rs
fn main() {
  let max_power = 10;
  let mut power = 1;
    while power < max_power {
        print!("{} ", power); // prints without newline
        power += 1;           // increment counter
    }
}
```

This prints the following output:

```
1 2 3 4 5 6 7 8 9
```

To start an infinite loop, use the `loop` statement, as shown below:

```
loop {
        power += 1;
        if power == 42 {
            // Skip the rest of this iteration
        continue;
        }
        print!("{}  ", power);
        if power == 50 {
           print!("OK, that's enough for today");
        break;  // exit the loop
        }
}
```

All the `power` values including `50` but except `42` are printed; then the loop stops with the statement `break`. The value `42` is not printed because of the `continue` statement. So, `loop` is equivalent to a `while true` and a `loop` with a conditioned `break` simulates a `do while` in other languages.

When loops are nested inside each other, the `break` and `continue` statements apply to the immediate enclosing loop. Any loop statement (also the `while` and `for` loop which we'll see next) can be preceded by a label (denoted as `labelname:`) to allow us to jump to the next or outer enclosing loop, as in this snippet:

```
'outer: loop {
      println!("Entered the outer dungeon.");
      inner: loop {
          println!("Entered the inner dungeon.");
          // break;    // this would break out of the inner loop
          break 'outer; // breaks to the outer loop
      }
      println!("This treasure can sadly never be reached.");
  }
  println!("Exited the outer dungeon!");
```

This prints the following output:

```
Entered the outer dungeon.
Entered the inner dungeon.
Exited the outer dungeon!
```

Obviously, the use of labels makes reading the code more difficult, so use it wisely. The infamous `goto` statement from C luckily does not exist in Rust!

Looping where a variable `var` begins from a start value `a` to an end value `b` (exclusive) is done with a `for` loop over a range expression, as shown in the following statement:

```
for var in a..b
```

Here is an example which prints the squares of the numbers `1` to `10`:

```
for n in 1..11 {
      println!("The square of {} is {}", n, n * n);
}
```

In general, `for in` loops over an iterator, which is an object that gives back a series of values one by one. The range `a..b` is the simplest form of iterator.

Each subsequent value is bound to the variable `n` and used in the next loop iteration. The `for` loop ends when there are no more values and the variable `n` then goes out of scope. If we don't need the value of the variable `n` in the loop, we can replace it with an _ (underscore), like this:

```
for _ in 1..11 { }
```

The many bugs in C-style `for` loops, like the off-by-one error with the counter, cannot occur here, because we loop over an iterator.

Variables can also be used in a range, like in the following snippet which prints nine dots:

```
let mut x = 10;
for _ in 1..x { x -= 1; print!("."); }
```

We'll examine iterators in more detail in `Chapter 5`, *Higher Order Functions and Error-Handling*.

Functions

The starting point of every Rust program is itself a function `fn` called the `main()` function, which can be further subdivided into separate functions for reuse of code or better code organization. Rust doesn't care in which order these functions are defined, but it is nice to put the function `main()` at the start of the code to get a better overview. Rust has incorporated many features of traditional functional languages; we will see examples of that in `Chapter 5`, *Higher Order Functions and Error-Handling*.

Let's start with a basic function example:

```
// from Chapter 3/code/functions.rs
fn main() {
  let hero1 = "Pac Man";
  let hero2 = "Riddick";
  greet(hero2);
  greet_both(hero1, hero2);
}

fn greet(name: &str) {
  println!("Hi mighty {}, what brings you here?", name);
}

fn greet_both(name1: &str, name2: &str) {
  greet(name1);
  greet(name2);
}
```

This prints out the following output:

```
Hi mighty Riddick, what brings you here?Hi mighty Pac Man, what brings you
here?Hi mighty Riddick, what brings you here?
```

Like variables, functions have variable `snake_case` names that must be unique, and their parameters (which have to be typed) are separated by commas, as in this example:

```
name1: &str, name2: &str
```

It looks like a binding, but without the `let` binding. Forcing a type to the parameters was an excellent design decision as it documents the function for use by its caller code and allows type inference inside the function. The type here is `&str` because strings are stored on the heap (see section *The stack and the heap* in `Chapter 2`, *Using Variables and Types*).

The functions above don't return anything useful (in fact, they return the unit value `()`), but, if we want a function to actually return a value, its type must be specified after an arrow `->`, as in this example:

```
fn increment_power(power: i32) -> i32 {
  println!("My power is going to increase:");
  power + 1
}

fn main() {
let power = increment_power(1); // function is called
println!(" I am now at power level: {}", power);}
```

When executed, it prints an output like the following:

```
My power is going to increase:
I am now at power level: 2
```

The return value of a function is the value of its last expression. Note that in order to return a value, the final expression must not end with a semicolon. What happens when you do end it with a semicolon? Try it out: in this case the unit value `()` is returned, and the compiler gives you the following error:

```
error: not all control paths return a value
```

We could have written the statement `return power + 1` as the last line, but that is not idiomatic code. If we wanted to return a value from the function before the last code line, we would have to write `return value;` as in the following:

```
if power < 100 { return 999 }
```

If this was the last line in the function, you would write it as follows:

```
if power < 100 { 999 }
```

A function can return only one value, but this isn't much of a limitation. If we have for example, three values, `a`, `b`, and `c`, to return, make one tuple `(a, b, c)` with them and return that. We will examine tuples in more detail in the next chapter.

A function that never returns is called a diverging function and it has return type `!`.

For example:

```
fn diverges() -> ! {
    panic!("This function never returns!");
}
```

It can be used as any type, for example to isolate exception handling, like in this example.

A function can be recursive; that means that the function calls itself, like in the following example:

```
// from Chapter 3/code/fib_procedural.rs
fn main() {
  let ans = fib(10);
  println!("{}", ans);
}

fn fib(x: i64) -> i64 {
     if x == 0 || x == 1 { return x; }
```

```
        fib(x - 1) + fib(x - 2)
}
```

Make sure that the recursion stops by including a base case, in this example when the function is called for `x` equal to `1` and `0`.

Functions have a type, for example, the type of the function `increment_power` from the previous code snippet like:

```
Fn(i32) -> i32
```

The `fn` function in general denotes a function type.

In Rust, you can also write a function inside another function (called a nested function), contrary to C or Java. This should only be used for small helper functions needed locally.

As an exercise, try the following:

Knowing that `if` can be an expression, simplify the following function:

```
fn verbose(x: i32) -> &'static str {
  let result: &'static str;
  if x < 10 {
    result = "less than 10";
  } else {
    result = "10 or more";
  }
  return result;
}
```

See code in `Chapter 3\exercises\ifreturn.rs`.

The `static in` and `static str` variables are a so-called lifetime indication, needed here to indicate how long the function's return value will exist. The static lifetime is the longest possible lifetime, such an object stays alive throughout the entire application, and so it is available in all of its code. We'll discuss this in detail in the section *Lifetimes* in `Chapter 6`, *Using Variables and Types*.

What is wrong with this function that returns the absolute of a given number in the variable `x`?

```
fn abs(x: i32) -> u32 {
   if x > 0 {
         x
   } else {
```

```
            -x
        }
    }
```

Correct and test it (see the code in `Chapter 3/exercises/absolute.rs`).

Documenting a function

Let's show an example of documenting code. In the `exdoc.rs` file, we have documented a function `cube` as follows:

```
fn main() {
  println!("The cube of 4 is {}", cube(4));
}
/// Calculates the cube `val * val * val`.
///
/// # Examples
///
/// ```
/// let cube = cube(val);
/// ```
pub fn cube(val: u32) -> u32 {
    val * val * val
}
```

If we now invoke `rustdoc exdoc.rs` on the command line, a `doc` folder is created. For a project do `cargo.doc` in the project's root folder. This contains a subfolder `exdoc`, with an `index.html` file that is the starting point of a website providing a documentation page for each function. For example, `fn.cube.html` shows:



Clicking on the `exdoc` link returns you to the index page.

Documentation comments are written in *Markdown* (for a brief intro, see `https://en.wikipedia.org/wiki/Markdown`). They can contain the following special sections preceded by a `#`. The examples are Panics, Failures, and Safety. Code appears between ```. For a function to be documented, it must belong to the public interface, so it must be prefixed with `pub`. See `Chapter 8`, *Organizing Code and Macros*.

A module can be documented with `//!` comments that start after the initial `{`.



For more info see
`https://doc.rust-lang.org/book/first-edition/documentation.html`.

Attributes

In the compiler, you may have already seen examples of warnings between `#[ ... ]` signs, like `#[warn(unused_variables)]`. These are attributes which represent metadata information about the code. You can use these yourself in code, and they are placed right before an item (such as a function) on which they have something to say. They can, for example, disable certain classes of warnings, turn on certain compiler features, or mark functions as being part of unit tests or benchmark code.

Conditional compilation

If you want to make a function that only works on a specific operating system, then annotate it with the `#[cfg(target_os = "xyz")]` attribute (where `xyz` can be one of `"windows"`, `"macos"`, `"linux"`, `"android"`, `"freebsd"`, `"dragonfly"`, `"bitrig"` or `"openbsd"`). For example, the following code works fine and runs on Windows:

```
   // from Chapter 3/code/attributes_cfg.rs
fn main() {
  on_windows();
}

#[cfg(target_os = "windows")]
fn on_windows() {
    println!("This machine has Windows as its OS.")
}
```

This produces the following output:

```
This machine has Windows as its OS.
```

If we try to build this code on a Linux machine, we get the following error:

```
error: unresolved name `on_windows
```

The code does not even build on Linux, because the attribute prevents it! Furthermore, you can even make your own custom conditions, see `http://rustbyexample.com/attribute/cfg/custom.html`.

Attributes are also used when testing and benchmarking code.

Testing

We can prefix a function with the `#[test]` attribute to indicate that it is part of the unit tests for our application or library. We then compile with the following command and run the generated executable:

```
rustc --test program.rs
```

This will replace the `main()` function with a test runner, and show the result from the functions marked with `#[test]`, for example:

```
// from Chapter 3/code/attributes_testing.rs
fn main() {
  println!("No tests are compiled,compile with rustc --test! ");
}

#[test]
fn arithmetic() {
  if 2 + 3 == 5 {
    println!("You can calculate!");
  }
}
```

Test functions, like `arithmetic()` in the example, are black boxes, they have no arguments or returns. When this program is run on the command line it produces the following output:

```
running 1 test
test arithmetic ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured
```

But, if we change the test to `if 2 + 3 == 6` the test passes as well! Try it out. It turns out that test functions always pass when their execution does not cause a crash (called a panic in Rust terminology), and only fails when it panics. That's why testing (or debugging) must use the `assert_eq!` macro (or other similar macros), as shown in the following code:

```
assert_eq!(2, power);
```

This statement tests whether the variable `power` has the value `2`. If it does, nothing happens, but if `power` is different from `2`, an exception occurs and the program panics, giving the following command:

```
thread '<main>' panicked at 'assertion failed.
```

In our first function, we would write the test `assert_eq!(5, 2 + 3);`, which would pass.

We could also write this as `assert!(2 + 3 == 5);` using the `assert!` macro. This macro does nothing if the expression between the parentheses is true, but it panics if the expression is false.

These macros are also useful in normal code, to make certain specific conditions are met. Just be aware that when they fail, they do it while the program is running!

A test fails when the function panics, as is the case for the following example:

```
#[test]
fn badtest() {
  assert_eq!(6, 2 + 3);
}
```

This produces the following output:

```
running 2 tests
test arithmetic ... ok
test badtest ... FAILED

failures:

---- badtest stdout ----
        thread 'badtest' panicked at 'assertion failed: `(left == right)` (left
 `6`, right: `5`)', attributes_testing.rs:21
note: Run with `RUST_BACKTRACE=1` for a backtrace.


failures:
    badtest

test result: FAILED. 1 passed; 1 failed; 0 ignored; 0 measured
```

If you want to make sure that a test fails, use the `#[should_panic]` attribute, like so:

```
#[test]
#[should_panic(expected = "assertion failed")]
fn failing_test() {
    assert!(6 == 2 + 3);
}
```

In this case `failing_test` passes OK, because this was what we expected! We'd better add the text `expected = assertion failed` to be sure the panic is caused by the assertion failing.

You can disable a test by giving it the additional `#[ignore]` attribute.

Passing tests are indicated in green, failing tests in red.

Unit test your functions by comparing the actual function result to the expected result with the macro call `assert_eq!(actual, expected)`. So, consider a function that looks like the following:

```
pub fn double(n: i32) -> i32 {
    n * 2
}
```

It is tested, as follows:

```
assert_eq!(double(42), 84);
```

The `pub` indicates that `double` is a public method, that can be called by client code using our library. Ordinary private methods should not be tested explicitly, they should be checked by calling public methods that test them.

If you compile without the test attribute, like the following command:

```
rustc attributes_testing.rs
```

No test functions are compiled and when run the `main()` function executes, which in our case prints the following output:

```
No tests are compiled, compile with rustc --test!
```

No test code is included in a normal build.

In a real project, the tests will be put in a separate `tests` module, as is done in next section (see `Chapter 8`, *Organizing Code and Macros*).

Testing with cargo

An executable project, or crate as it is called in Rust, needs to have a startup function `main()`, but a library crate, to be used in other crates, does not need a function `main()`. Create a new library crate `mylib` with `cargo` as follows:

```
cargo new mylib
```

This creates a subfolder `src` with a source file `lib.rs` which contains the following:

```
#[cfg(test)]
mod tests {
    #[test]
    fn it_works() {
    }
}
```

So, a library crate is created with no code of its own, but it does contain a test template annotated with a `cfg(test)` attribute. This attribute indicates that the code that follows will only be compiled in test mode. To differentiate with normal library code, use a prefix not in the attribute like this:

```
#[cfg(not(test))]
fn main() {
    println!("Normal mode, no test was compiled");
}
```

In the test section you can add the unit tests you write on the functions of your library. To run these tests, go to the project root folder and type `cargo test`, which produces similar output as in the previous section.

You can run a single test by supplying its function name, like this:

```
cargo test it_works
```

The command `cargo test` runs tests in parallel whenever possible. If this can pose a problem, perhaps because one test depends on another, you can execute them all in one thread using the following command:

```
cargo test – – – –test-threads=1
```

The tests module

In a more realistic, larger project, tests are separated from the application code:

- Unit tests are collected in a module `test`
- Integration tests are collected in a `lib.rs` file in a `tests` directory

The code generated by Cargo for a library groups the tests inside a `mod tests`, a so-called module (see `Chapter 8`, *Organizing Code and Macros;* if you prefer, you can come back to this section after having read `Chapter 8`, *Organizing Code and Macros* and have a better understanding of modules).

In order to work with functions defined in the main code, we have to add the command `use super::*;`, which brings all these functions into the scope of the tests module:

```
pub fn double(n: i32) -> i32 {
    n * 2
}

#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    fn it_works() {
        assert_eq!(double(42), 84);
    }
}
```

The module `tests` is typically used to contain unit tests of your library's functions.

Let's make another example with integration tests using our function `cube` from `Chapter 3`, *Using Functions and Control Structures*, and start a project with `cargo new cube`. We replace the code in `src\lib.rs` with:

```
// from Chapter 3/code/cube/src/lib.rs:

pub fn cube(val: u32) -> u32 {
    val * val * val
  }

#[cfg(test)]
mod tests;
```

In the second line, we declare our `tests` module, preceded by the test configuration attribute. Now the code of this module goes into a file `tests.rs` in the same folder, that way they are more cleanly separated from our library code:

```
// from Chapter 3/code/cube/src/tests.rs:
use super::*;

#[test]
fn cube_of_2_is_8() {
    assert_eq!(cube(2), 8);
}

// other test functions:
//...
```

Integration tests go into a file `lib.rs` in a `tests` folder, which we create manually:

```
// from Chapter 3/code/cube/tests/lib.rs:
extern crate cube;
use cube::cube;

#[test]
fn cube_of_4_is_64() {
    assert_eq!(cube(4), 64);
}

// other test functions:
// ...
```

Here, we need to import the `cube` crate with an `extern` command, and qualify the function name `cube` with its module name `cube` (or else do a `use cube::cube;`).

As before, the test code will only be compiled and run when we give the `cargo test` command, which gives the following results:

```
ntrol structures\code\cube>cargo test
   Compiling cube v0.0.1 (file:///F:/Rust/2ndeditionRE/Rust_Essentials/C
Chapter%203%20-%20Using%20functions%20and%20control%20structures/code/cu
    Finished dev [unoptimized + debuginfo] target(s) in 1.21 secs
     Running target\debug\deps\cube-a9231f47f21e9d01.exe

running 1 test
test tests::cube_of_2_is_8 ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured

     Running target\debug\deps\lib-d0a819c57658b564.exe

running 1 test
test cube_of_4_is_64 ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured

   Doc-tests cube

running 0 tests

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

We see that our two tests (the unit test and the integration test) passed. The output shows at the end that tests in the documentation are also executed if they are present.

If you want to be able to use a more Speclike framework, with keywords like **describe** and **it**, you should definitely take a look at the stainless crate (`https://github.com/reem/stainless`).

Summary

In this chapter, we learned to make basic programs by using `if` conditions, `while`, and `for` loops, and using functions to structure our code. Lastly, we saw the immense power attributes give to widen Rust's possibilities, and we applied this in conditional compilation and testing.

In the following chapter, we start using composite values and explore the powers of pattern matching.

4

Structuring Data and Matching Patterns

Until now we only used simple data, but to do real programming, more composite and structured data values are needed. Amongst them are flexible arrays and tuples, enums, and structs to represent more object-like behavior, like in classical object-oriented languages. Options are another important type used to ensure that cases where no value is returned are accounted for. Then we look at pattern matching, another typical functional construct in Rust. But we start by looking more carefully at strings.

We will cover the following topics:

- Strings
- Arrays, vectors and slices
- Tuples
- Structs
- Enums
- Getting input from the console
- Matching patterns
- Program arguments

Strings

The way Rust works with strings differs a bit from strings in other languages. All strings are valid sequences of Unicode (UTF8) bytes. They can contain null bytes, but they are not null terminated as in C.

Rust distinguishes two types of string:

- The strings we have used until now are **string slices**, whose type is `&str`. The `&` points out that a string slice is a reference to a string. They are immutable and have a fixed size. For example, the following bindings declare string slices:

```
// from Chapter 4/code/strings.rs
let magician1 = "Merlin";
let greeting = "Hello, world!";
```

- Or if we explicitly annotate the string variable with its type:

```
let magician2: &str = "Gandalf";
```

- We can also define it as a string literal:

```
let magician2: &'static str = "Gandalf";
```

- The `&'static` denotes that the string is statically allocated, and stored directly in the executable program. When declaring global constants, indicating the type is mandatory, but for a let binding it is optional because the compiler infers the type.

```
println!("Magician {} greets magician {} with {}",
magician1, magician2, greeting);
```

- This gives the following output:

```
Magician Merlin greets magician Gandalf with Hello, world!
```

- Literal strings live as long as the program; they have the lifetime of the program, which is the `static` lifetime. They are defined in the `std::str` module.

- A String on the other hand can grow dynamically in size (in fact it is a buffer), and so it must be allocated on the heap. We can create an empty string with:

```
let mut str1 = String::new();
```

- Each time the string grows it has to be reallocated in memory. So if you know it will start out as 25 bytes for example, you can create the string with this amount of memory allocated as:

```
let mut str2 = String::with_capacity(25);
```

- As long as the size of `str2` is less than 25, there is no reallocation when adding to it. This type is described in the module `std::string`.

- To convert a string slice into a string, use the method `to_string`:

```
let mut str3 = magician1.to_string();
```

The method `to_string()` can be used to convert any object to a string (more precisely; any object that implements the `ToString` trait; we will talk about traits in the next chapter) and this method needs to allocate memory on the heap.

If `str3` is a string, then you can make a string slice from it with `&str3`:

```
let sl1 = &str3;
```

A string slice created this way can be considered as a view into the string. It is a reference to the interior of the string, and making it has no cost.

This way is preferable to `to_string()` when comparing strings, because using `&` doesn't consume resources, while the function `to_string()` allocates heap memory:

```
if &str3 == magician1 {
    println!("We got the same magician alright!")
}
```

String slices can also be created with a range notation like `&str3[1..3]`. In fact `&str3[..]` is the same as `&str3`.

To build up a string, we can use a number of methods:

- `push`: To append a character to the string
- `push_str`: To append another string to the string

You can see them in action in this code snippet:

```
let c1 = 'Θ';  // character c1
str1.push(c1);
println!("{}", str1); // Θ
str1.push_str(" Level 1 is finished - ");
```

```
println!("{}", str1); // Θ Level 1 is finished –
str1.push_str("Rise up to Level 2");
println!("{}", str1); // Θ Level 1 is finished – Rise up to Level 2
```

If you need to get the characters of a string one by one and in order, use the method `chars()`. This method returns an `Iterator`, so we can use the `for in` loop (see the section *Looping* in `Chapter 2`, *Using Variables and Types*):

```
for c in magician1.chars() {
    print!("{} – ", c);
}
```

This prints the output as:

```
M – e – r – l – i – n –
```

To loop over the parts of a string separated by whitespace, we can use the method `split()`, which also returns an `Iterator`:

```
for word in str1.split(" ") {
    print!("{} / ", word);
}
```

This prints out the following output:

```
Θ / Level / 1 / is / finished / – / Rise / up / to / Level / 2 /
```

To change the first part of a string that matches with another string, use the `replace`:

```
let str5 = str1.replace("Level", "Floor");
```

This code allocates new heap memory for the modified string `str5`.

It prints out the following output:

```
Floor 1 is finished – Rise up to Floor 2
```

Strings can be concatenated with the + operator, but the second argument needs to be of the type string slice `&str`:

```
let st1 = "United ".to_string(); // st1 is of type String
let st2 = "States";
let country = st1 + st2;
println!("The country is {}", country);
let st3 = "United ".to_string();
let st4 = "States".to_string();
let country = st3 + &st4;
println!("The country is {}", country);
```

This gives the following output:

```
The country is United States
The country is United States
```

When writing a function that takes a string as argument, always declare it as a string slice, which is a view into the string, like this:

```
fn how_long(s: &str) -> usize {  s.len() }
```

The reason is that passing a string `str1` as an argument allocates memory, and passing it as a slice does not allocate. The easiest way to do this is the following:

```
println!("Length of Merlin: {}", how_long(magician1));
println!("Length of str1: {}", how_long(&str1));
```

You can also pass a part of the string with the range notation, or you can even do the following:

```
println!("Length of str1: {}", how_long(&str1[3..4]));
```

The preceding statements print out the following output:

```
Length of Merlin: 6
Length of str1: 43
Length of str1: 4
```

Raw strings are prefixed with `r` or `r#`. They are printed out literally - new lines or escaped characters are not interpreted, like this:

```
println!("{}", "Ru\nst");
println!("{}", r"Ru\nst");
println!("{}", r#"Ru\nst"#);
```

This prints out the following output:

```
Ru\nst
Ru\nst
Ru\nst
```

To convert numbers into strings, use the method `from_str`, illustrated here:

```
println!("{:?}", f64::from_str("3.6"));
let number: f64 = f64::from_str("3.6").unwrap();
```

The `from_str` method also needs the trait `FromStr`; to use it, you must import the trait with `use`, as follows:

```
use std::str::FromStr;
```

Consult the docs (`http://doc.rust-lang.org/std/str/` and `http://doc.rust-lang.org/std/string/`) for more functionality.

Here is a schema to better see the difference between the two string types:

String	**String slice (`&str`)**
Mutable: heap memory allocation Module: `std::string`	Fixed size: view on string reference(`&`) Module: `std::str`

Arrays, vectors, and slices

Suppose we have a bunch of alien creatures to populate a game level, then we probably want to store their names in a handy list. Rust's array is just what we need:

```
// from Chapter 4/code/arrays.rs
let aliens = ["Cherfer", "Fynock", "Shirack", "Zuxu"];
println!("{:?}", aliens);
```

To make an array, separate the different items by commas and surround the whole with `[ ]` (rectangular brackets). All items must be of the same type. Such an array is of fixed size (it must be known at compile time) and cannot be changed; it is stored in one contiguous piece of memory in stack memory.

If the items have to be modifiable, declare your array with `let mut`; however even then the number of items cannot change:

```
let mut aliens = ["Cherfer", "Fynock", "Shirack", "Zuxu"];
```

The array `aliens` could be type annotated as `[&str; 4]`, where the first parameter is the type of the items, and the second is their number.

```
let aliens: [&str; 4] = ["Cherfer", "Fynock", "Shirack", "Zuxu"];
```

If we want to initialize an array with three `Zuxus`, that's easy too:

```
let zuxus = ["Zuxu"; 3];
```

How would you make an empty array? Like this:

```
let mut empty: [i32; 0] = [];
println!("{:?}", empty); // []
```

We can access individual items with their index, starting from `0`:

```
println!("The first item is: {}", aliens[0]); // Cherfer
println!("The third item is: {}", aliens[2]); // Shirack
```

The number of items in the array is given by `aliens.len()`, so how would you get the last item?

Using the following statement:

```
aliens[aliens.len() - 1]
```

Alternatively, this can be found with the following statement:

```
aliens.iter().last().unwrap();
```

What do you think will happen when we try to change an item, like this?

```
aliens[2] = "Facehugger";
```

Hopefully you didn't think that Rust would allow this, did you? Unless you told it explicitly that `aliens` can change with the following statement:

```
let mut aliens = [...];
```

In many cases the compiler checks whether the array index stays within the array bounds.

The index is also checked at runtime to be within the array bounds `0` and `aliens.len();` if it is not, the program crashes with a runtime error or panic:

```
println!("This item does not exist: {}", aliens[10]);
```

This gives the following output:

```
thread '<main>' panicked at 'index out of bounds: the len is 4 but the
index is 10'
```

Pointers to arrays use automatic dereferencing so that you do not need to use `*` explicitly, as demonstrated in this code snippet:

```
let pa = &aliens;
println!("Third item via pointer: {}", pa[2]);
```

This prints the following output:

```
Third item via pointer: Shirack
```

If we want to go through the items successively one by one, and print them out or do something useful with them, we can do it like this:

```
for ix in 0..aliens.len() {
    println!("Alien no {} is {}", ix, aliens[ix]);
}
```

This works and it gives us for each item its index, which might be useful. But we use the index to fetch each consecutive item, and each time Rust has to check whether we are still within the bounds of the array in memory. That's why it is not very efficient, and in the section *Iterators* in `Chapter 6`, *Using Traits and OOP in Rust*, we will see a much more efficient way by iterating over the items:

```
for a in aliens.iter() {
    println!("The next alien is {}", a);
}
```

The `for` loop can be written even shorter as:

```
for a in &aliens  { ... }
```

Vectors

Often it is more practical to work with a kind of array than can grow (or shrink) in size because it is allocated on the heap. Rust provides this through the vector type `Vec`, from the module `std::vec`. This is a generic type, which means that the items can have any type `T`, where `T` is specified in the code; for example we can have `Vec<i32>` or `Vec<&str>` vector types. To indicate that it is a generic type, it is written as `Vec<T>`, where all elements must be of the same type `T`.

We can make a vector in either of two ways, with the function `new()` or with the macro `vec!`:

```
let mut numbers: Vec<i32> = Vec::new();
let mut magic_numbers = vec![7i32, 42, 47, 45, 54];
```

In the first case the type is indicated explicitly with the vector type `Vec<i32>`, in the second case this is done by giving the first item an `i32` suffix, but this is usually optional.

We can also make a new vector with an initial memory size allocated to it, which can be useful if you know in advance that you will need at least that many items. The following initializes a vector for signed integers, with memory allocated for `25` integers:

```
let mut ids: Vec<i32> = Vec::with_capacity(25);
```

We need to provide the type here, otherwise the compiler can not calculate the amount of memory needed.

A vector can also be constructed from an `Iterator` through the method `collect()`, like in this example with a range:

```
let rgvec: Vec<u32> = (0..7).collect();
println!("Collected the range into: {:?}", rgvec);
```

This prints the following output:

```
Collected the range into: [0, 1, 2, 3, 4, 5, 6]
```

Indexing, getting the length and looping over a vector works the same as with arrays. For example, a `for` loop over a vector can be written simply as:

```
let values = vec![1, 2, 3];
for n in values {
    println!("{}", n);
}
```

This prints out the following output on consecutive lines:

```
1 2 3
```

Add a new item to the end of a vector with the function `push()`, and remove the last item with the function `pop()`.

```
numbers.push(magic_numbers[1]);
numbers.push(magic_numbers[4]);
println!("{:?}", numbers); // [42, 54]
let fifty_four = numbers.pop();// fifty_four now contains 54
println!("{:?}", numbers); // [42]
```

If you need to mutate the vector while iterating, use `iter_mut`:

```
let mut vec = vec![1, 2, 3, 4];
for x in vec.iter_mut() {
    *x += 1;
}
println!("Mutated vector: {:?}", vec);
```

This prints out the following output:

```
Mutated vector: [2, 3, 4, 5]
```

The `x` variable here is a reference to the successive items of the vector, to obtain its value you must use the dereferencing operator `*`.

If you omit the `*`, you get the following error:

```
error[E0368]: binary assignment operation '+=' cannot be applied to type
'&mut {integer}'
```

If a function needs to return many values of the same type, make an array or vector with these values and return that object.

Slices

What if you want to do something with a part of an array or vector? Perhaps your first idea is to copy that part out to another array, but Rust has a safer and more efficient solution: take a slice of the array. No copy is needed, instead you get a view into the existing array, like a string slice is a view into a string.

As an example, suppose I only need the numbers `42`, `47`, and `45` from our vector `magic_numbers`. Then I can take the following slice:

```
let slc = &magic_numbers[1..4]; // only the items 42, 47 and 45
```

The starting index `1` is the index of `42`, the last index `4` points to `54`, but this item is not included. The `&` shows that we are referencing an existing memory allocation.

Slices share the following with vectors:

- They are generic and have type `&[T]` for a type `T`
- Their size does not have to be known at compile time

Strings and arrays

Back in the first section of this chapter we saw that the sequence of characters in a string is given by the function `chars()`. Doesn't this look like an array to you? A string is backed up by an array if we look at the memory allocation of its characters; it is stored as a vector of bytes `Vec<u8>`.

That means we can also take a slice of type `&str` from a string:

```
let location = "Middle-Earth";
let part = &location[7..12];
println!("{}", part); // Earth
```

We can collect the characters of a string slice into a vector and sort them as follows:

```
let magician = "Merlin";
let mut chars: Vec<char> = magician.chars().collect();
chars.sort();
for c in chars.iter() {
    print!("{} ", c);
}
```

This prints out the following output:

```
M e i l n r
```

The capital letters come before small letters in the sort order.

Here are some other examples of using the method `collect()`:

```
let v: Vec<&str> = "The wizard of Oz".split(' ').collect();
```

The variable `v` then contains, `["The", "wizard", "of", "Oz"]`.

```
let v: Vec<&str> = "abc1def2ghi".split(|c: char| c.is_numeric()).collect();
```

The variable `v` is split on the integers and now contains, `["abc", "def", "ghi"]`.

Here `split()` takes a closure to determine on which character to split.

To convert from a `&str` to a `&[u8]` use `as_bytes()`:

```
let arr = magician.as_bytes();
println!("{:?}", arr ); // [77, 101, 114, 108, 105, 110]
```

Both slice types, `&str` and `&[T]`, can be seen as views into strings and vectors respectively. The following schema compares the types we have just encountered (`T` denotes a generic type):

fixed-size (stack-allocated)	slices	are view into	dynamic size (growable) (heap-allocated)
String slice `&str`	Type: `&[u8]`		String
Array Type: `[T; size]`	Type: `&[T]`		Vector-type: `Vec<T>`

Exercises:



(see `Chapter 4/exercises/chars_string.rs`):
1. Try out whether you can get the first or the fifth character of a string by using `[0]` or `[4]`.
2. Compare the `bytes()` method with `chars()` on the string `;let greeting = "Hello ;world!";;`

Tuples

If you want to combine a certain number of values of different types, then you can collect them in a tuple, enclosed between parentheses (`( )`) and separated by commas, like this:

```
// from Chapter 4/code/tuples.rs
let thor = ("Thor", true, 3500u32);
println!("{:?}", thor); // ("Thor", true, 3500)
```

The type of `thor` is `(&str, bool, u32)`, that is, the tuple of the item's types.

To extract an item on index use a dot syntax:

```
println!("{} - {} - {}", thor.0, thor.1, thor.2);
```

Another way to extract items to other variables is by destructuring the tuple:

```
let (name, _, power) = thor;
println!("{} has {} power points ", name, power);
```

This prints out the following output:

```
Thor has 3500 power points
```

Here the `let` statement matches the pattern on the left with the right-hand side. The _ indicates that we are not interested in the second item of `thor`.

Tuples can only be assigned to one another or compared with each other if they are of the same type. A one-element tuple needs to be written like this:

```
let one = (1,);
```

A function that needs to return some values of different types can collect them in a tuple and return that tuple, like this:

```
fn increase_power(name: &str, power: u32)
-> (&str, u32) {
if power > 1000 {
 return (name, power * 3);
} else {
 return (name, power * 2);
 }
}
```

This function header could also be written as:

```
fn increase_power2((name, power): (&str, u32)) -> (&str, u32)
```

If we call this with the following statement:

```
let (god, strength) = increase_power(thor.0, thor.2);
println!("This god {} has now strength {} ", god, strength);
```

The output looks like the following:

```
This god Thor has now strength 10500
```

A tuple is also very handy when swapping two variables:

```
let mut n = 0;
let mut m = 1;
let (n, m) = (m, n);
println!("n: {} m: {}", n, m);
```



Exercise:
(see code in `Chapter 4/exercises/tuples_ex.rs`)
Try to compare the tuples `(2, 'a')` and `(5, false)`, explain the error message.
Try to change an item in a tuple (hint: you must add a keyword for it to work).
Make an empty tuple.
Haven't we encountered this before? So the unit value is in fact an empty tuple!

Structs

Often you need to keep several values of possibly different types together in your program, for example, the scores of the players. Let us say that the `score` contains numbers indicating the `health` of the players and the level at which they are playing. The first thing you can do to clarify your code is to give these tuples a common name, like:

```
struct Score;
```

Or better still, indicate the types of the values:

```
struct Score(i32, u8);
```

And we can make a score like this:

```
// from Chapter 4/code/structs.rs
let score1 = Score(73, 2);
```

These are called **tuple structs** because they resemble tuples very much.The values contained in them can be extracted like this:

```
let Score(h, l) = score1; // destructure the tuple
println!("Health {} - Level {}", h, l);
```

This prints the following output:

```
Health 73 - Level 2
```

A tuple struct with only one field (called a **newtype)** gives us the possibility to create a new type based on an old one, so that both have the same memory representation. Here is an example:

```
struct Kilograms(u32);
let weight = Kilograms(250);
```

```
let Kilograms(kgm) = weight; // extracting kgm
println!("weight is {} kilograms", kgm);
```

This prints the following output:

```
weight is 250 kilograms
```

But with these structs we still have to remember what these numbers mean, and to which player they belong. We can make coding much simpler by defining a struct with **named fields**:

```
struct Player {
    nname: &'static str, // nickname
    health: i32,
    level: u8
}
```

This could be defined inside `main()` or outside of it, which is the preferred way.

Now we can make `Player` instances or objects like this:

```
let mut pl1 = Player{nname: "Dzenan", health: 73, level: 2};
```

Note the `{ }` (curly braces) around the object, and the key value syntax. The field `nname` is a constant string, and Rust requires that we indicate its lifetime, that is how long this string will be needed in the program. We used the global scope.

We can access the fields of the instance with the dot notation:

```
println!("Player {} is at level {}", pl1.nname, pl1.level);
```

This produces the following output:

```
Player 1 Dzenan is at level 2
```

The variable ;`struct` has to be declared as mutable if the field values can change, for example, when the player enters a new level:

```
pl1.level = 3;
```

By convention the name of a `struct` always starts with a capital letter and follows *CamelCase*. It also defines a type of its own, composed of the types of its items.

You can define a new instance `struct` starting from an existing one like this:

```
let pl2 = Player{ nname: "Ivo", ..pl1 };
println!("Player 2 {} is at level {}", pl2.nname, pl2.level);
```

This produces the following output:

```
Player 2 Ivo is at level 3
```

Like tuples, structs can also be destructured in a `let` binding, for example:

```
let Player{health: ht, nname: nn, ..} = pl1;
println!("Player {} has health {}", nn, ht);
```

This prints out the following output:

```
Player Dzenan has health 73
```

This shows that you can rename fields, reorder them if you want, or leave fields out with pointers indicated with `&`, and do automatic dereferencing when accessing data structure elements, like here:

```
let ps = &Player{ nname: "John", health: 95, level: 1 };
println!("{} == {}", ps.nname, (*ps).nname);
```

This prints out the following output:

```
John == John
```

The structs are quite similar to records, or structs in C or even classes in other languages. In `Chapter 6`, *Using Traits and OOP in Rust*, we will see how we can define methods on structs, in particular how to define a constructor `new` like in object-oriented languages.



Exercise:
(see code in `Chapter 4/exercises/monster.rs`)
Define a struct `Monster` with fields `health` and `damage`. Make a `Monster` and show its condition.

Enums

If something can only be one of a limited number of named values, then we define it as an `enum`. For example, if our game needs the compass directions, we could define:

```
// from Chapter 4/code/enums.rs
enum Compass {
   North, South, East, West
}
```

And use it like this in function `main()` or any another function:

```
let direction = Compass::West;
```

The `enum`'s values can also be other types or structs, like in this example, where we define a type `species`:

```
type species = &'static str;

enum PlanetaryMonster {
   VenusMonster(species, i32),
   MarsMonster(species, i32)
}

let martian = PlanetaryMonster::MarsMonster("Chela", 42);
```

The enums are sometimes called **union types** or **algebraic data types** in other languages.

If we make a `use` at the start of the code file:

```
use PlanetaryMonster::MarsMonster;
```

Then the type can be shortened, like this:

```
let martian = MarsMonster("Chela", 42);
```

The enums are really nice to bring readability in your code, and they are used a lot in Rust. To apply them usefully in code, see the following section on *Matching patterns*.

Result and Option

Here we look at two kinds of `enum` that are used everywhere in Rust code. A `Result` is a special kind of `enum` that is defined in the standard library. It is used whenever something is executed, that can either end:

- Successfully, then a value `Ok` (of a certain type `T`) is returned
- With an error, then an `Err` value (of type `E`) is returned

Because this situation is so common, provision is made that the value `T` and error `E` types can be as general, or generic, as possible. The `Result enum` is defined as:

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

An `Option` is another `enum` that is defined in the standard library. It is used whenever there is a value, but there can also be a possibility that there is no value. For example, suppose our program expects to read in a value from the console. However, when it runs as a background program by accident, it will never get an input value.

Rust wants to be on the safe side whenever possible, so in this case it is better to read in the value as an `Option` `enum`, with two possibilities:

- `Some`: If there is a value of type `T`
- `None`: If there is no value

The value `Option` again is defined as a generic type:

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

Getting input from the console

Suppose we want to capture the nickname of our `player`(s) before starting the game, how would we do that? Input or output functionality is handled by the module `io` in the crate `std`. It has a function `stdin()` to read input from the console. This function returns an object of type `Stdin`, which is a handle to the input stream. The object `Stdin` has a method `read_line(buf)` to read a full line of input, that ends with a new line character (that is, when the user hits *Enter*). This input is read into a `String` buffer `buf`. A method is a name for a function defined for a certain type, and it is called using dot notation, like `object.method` (see `Chapter 6`, ;*Using Traits and OOP in Rust*).

So our code will look like this:

```
let mut buf = String::new();
io::stdin().read_line(&mut buf);
```

But that is not good enough for Rust, it gives us the following warning:

```
warning: unused result which must be used
```

Rust is foremost a safe language and we must be ready to cope with everything than can occur. Reading a line might work if an input value is given. But it can also fail, for example, if this code was running in a background job on a machine, so that no console is available to get input from.

How to cope with that? Well, the function `read_line()` returns a value `Result`, that can either be a real value (an `Ok`), when everything works fine, or an error value (an `Err`), when there is a problem. To cope with a possible error, we need an `ok()` and an `expect()` function. The function `ok()` converts the value `Result` into a value `Option` (which contains how many bytes were read), the function `expect()` gives that value, or shows its message when an error occurs. In Rust a program panics when an error occurs that cannot be recovered from, and the string argument from the function `expect()` is displayed to tell us where it occurs.

This is written in Rust in a chained form (a bit unusual the first time you see it) as follows:

```
io::stdin().read_line(&mut buf).ok().expect("Error!");
```

Rust allows us to write these successive calls on separate lines, which for most people clarifies the code a lot:

```
// from Chapter 4/code/input.rs
use std::io;

fn main() {
   println!("What's your name, noble warrior?");
   let mut buf = String::new();
   io::stdin().read_line(&mut buf)
              .ok()
              .expect("Failed to read line");
   println!("{}, that's a mighty name indeed!", buf);
}
```

When running this code from the command line we get the following conversation:

```
What's your name, noble warrior?
Riddick
Riddick
that's a mighty name indeed!
```

Can you guess why `that's a mighty name indeed!` appears on a new line? Indeed, the input `buf` still contains a newline character `\n`! Luckily we have a method `trim()` to remove trailing and leading whitespace from a string. If we insert the line:

```
let name = buf.trim();
println!("{}, that's a mighty name indeed!", name);
```

We now get a correct output, like the following:

```
Riddick, that's a mighty name indeed!
```

In case the input does not succeed, our program crashes with the following output:

```
What's your name, noble warrior?
thread '<main>' panicked at 'Failed to read line'
```

Instead of using this chained `.ok().expect()` form, you can also use a shorter form with a function `unwrap()` like this:

```
io::stdin().read_line(&mut buf).unwrap();
```

This unwraps a value `Result`, yielding the content of a value `Ok` (or of a value `Some` in the case of a value `Option`), but panics if the value is an `Err` (or a `None` for `Option`), with a panic message provided by the `Err`'s value.

Another alternative is to test with the function `is_ok()`, which returns `true` if the result is `Ok`:

```
if io::stdin().read_line(&mut buf).is_ok() {
    let name = buf.trim();
    println!("{}, that's a mighty name indeed!", name);
}
else {
    println!("Failed to read line!");
}
```

How would we read in a positive integer number from the console?

```
// from Chapter 4/code/pattern_match.rs
let mut buf = String::new();
io::stdin().read_line(&mut buf)
               .ok()
               .expect("Failed to read number");
   let input_num: Result<u32, _> = buf.trim().parse();
```

We read the number in from the console in a `String` buffer `buf` and function `trim()` the value; the function `expect()` will show us the message if something goes wrong. But what we have read in this is still a `String`, we must convert that `String` to a number.

The method `parse()` tries to convert the input to an unsigned 32-bit integer in this case. What it returns is in fact again a value `Result`, this can either be an integer (`Ok<u32>`), or an error (`Err`) when the conversion fails.

We will encounter more examples of `Option` and `Result` in the section *Generics* in `Chapter 5`, *Higher Order Functions and Error-Handling*.

Matching patterns

But how will we test whether `input_num` from the previous section, which is of type `Result`, contains a value or not? When the value is an `Ok(T)`, the function `unwrap()` can extract the value `T`, like this:

```
println!("Unwrap found {}", input_num.unwrap());
```

This prints the following output:

```
    Unwrap found 42
```

But when the result is an `Err` value, this lets the program crash with a panic:

```
    thread '<main>' panicked at 'called `Result::unwrap()` on an `Err`
value'.
```

This is bad! To solve this, no complex `if - else` constructs will do; we need Rust's magical `match` here, which has a lot more possibilities than the case or switch in other languages:

```
// from Chapter 4/code/pattern_match.rs
match input_num {
   Ok(num) => println!("{}", num),
   Err(ex) => println!("Please input an integer number! {}", ex)
};
```

The `match` tests the value of an expression against all possible values. Only the code (which could be a block) after the arrow `=>` of the first matching branch is executed. All branches are separated by commas. In this case the same number that is given as input is printed out. There is no fall through from one branch to the next, so a break statement is not necessary, avoiding a common bug in C and C++.

The good thing about `match` is that it is exhaustive - all possible values must be present as branches. If we omit the `Ok` branch for example, we get a compiler error as follows: `error: pattern `Err(_)` not covered`.

In order to continue working with the return value of `match`, we have to bind that value to a variable, which is possible because `match` itself is an expression:

```
let num = match input_num {
    Ok(num) => num,
    Err(_)  => 0
};
```

This `match` extracts the number from `input_num`, so that we can compare it with other numbers or calculate with it. Both branches must return a value of the same type, that's why we returned `0` in the `Err` case (supposing we expect a number greater than `0`).

An alternative way to get the `Result` or `Option` value is using the `if let` construct, like this:

```
if let Ok(val) = input_num {
    println!("Matched {:?}!", val);
} else {
    println!("No match!");
}
```

The `input_num` is destructured and if it contains a value `val`, this is extracted.

The same principle can be applied inside a `while` loop, like this:

```
while let Ok(val) = input_num {
    println!("Matched {:?}!", val);
    if val == 42 { break }
}
```

If another value than `42` is given, this causes an infinite loop, which you can stop with *Ctrl+C*.

With `match`, all possible values must be covered, which is the case if we match on a `Result`, `Option` (`Some` or `None` is pretty exhaustive), or some other `enum` value.

But watch what happens when we test on a string slice:

```
// from Chapter 4/code/pattern_match2.rs
let magician = "Gandalf";
match magician {
    "Gandalf" => println!("A good magician!"),
    "Sauron"  => println!("A magician turned bad!")
}
```

This `match` on `magician` gives us the following error:

```
    error: non-exhaustive patterns: `_` not covered.
```

After all there are other magicians besides `Gandalf` and `Sauron`! The compiler even gives us the solution; use an underscore (_) for all other possibilities, so this is a complete match:

```
match magician {
    "Gandalf" => println!("A good magician!"),
    "Sauron"  => println!("A magician turned bad!"),
```

```
    _          => println!("No magician turned up!")
}
```

To always be on the safe side, prefer `match` ;to other constructs when testing on the possible values of a variable or expression!

The left-hand side of a branch can contain several values if they are separated by a pipe sign `|`, or an inclusive range of values written as `start ... end`. The following snippet shows this in action:

```
let magical_number: i32 = 42;
match magical_number {
      // Match a single value
       1 => println!("Unity!"),
      // Match several values
       2 | 3 | 5 | 7 | 11 => println!("Ok, these are primes"),
      // Match an inclusive range
       40...42 => println!("It is contained in this range"),
      // Handle the rest of cases
        _ => println!("No magic at all!"),
}
```

This prints out the following output:

```
It is contained in this range
```

The matched value can be captured in a variable (here `num`) using the `@` symbol, like this:

```
num @ 40...42 => println!("{} is contained in this range", num)
```

This prints the following output:

```
42 is contained in this range
```

The use of the `..` and `...` notations can be confusing, so here is a summary:

	What works	Does not work
`for in`	.. exclusive	...
`match`	... inclusive	..

Matching enums is very straightforward:

```
let direction = Compass::West;
match direction {
    Compass::North   => println!("Go to the North!"),
    Compass::East    => println!("Go to the East!"),
```

```
        Compass::South   => println!("Go to the South!"),
        Compass::West    => println!("Go to the West!"),
    }
```

This prints out the following output:

```
    Go to the West!
```

The `destructure_enum.rs` has more examples of destructuring enums:

```
use En::*;

struct St {
   f1: i32,
   f2: f32
}

enum En {
   Var1,
   Var2,
   Var3(i32),
   Var4(i32, St, i32)
}

fn foo(x: En) {
   match x {
      Var1 => println!("first variant"),
      Var2 => println!("second variant"),
      Var3(..) => println!("third variant"),
      Var4(..) => println!("fourth variant")
   }
}

fn foo2(x: &En) {
   match x {
      &Var1 => println!("first variant"),
      &Var2 => println!("second variant"),
      &Var3(5) => println!("third variant with number 5"),
&Var3(x) => println!("third variant with number {} (not 5)", x),
      &Var4(3, St{ f1: 3, f2: x }, 45) => {
println!("destructuring an embedded struct, found {} in f2", x)
      }
      _ => println!("other (Var4)")
      // cannot move out of borrowed content:
      // &Var4(_, v, _) => {
   //    println!("Some other Var4 with {} in f1 and {} in      // f2",
v.f1, v.f2)
      // }
```

```
        // // _ => println!("other (Var2)") // unreachable pattern
    }
}

fn main() {
    let en1 = En::Var3(42);
    // foo(en1); // third variant
    foo2(&en1); // third variant with number 42 (not 5)
    let st1 = St { f1: 3, f2: 10.0 };
    // let en2 = En::Var4(3, st1, 45);
    // foo2(&en2); // destructuring an embedded struct, found 10 in f2
    let en3 = En::Var4(3, st1, 42);
    foo2(&en3); // other (Var4)
}
// third variant with number 42 (not 5)
// other (Var4)
```

Here is an example of matching a tuple:

```
    // from Chapter 4/code/match_tuple.rs
fn main() {
    let status = (42, true);
    match status {
        (0, true) => println!("Success"),
    // Any first value matches in the following branch:
        (_, true) => println!("Pyrrhic victory"),
        // Any pair of values will match in the following branch:
        (_, _)    => println!("Complete loss")
    }
}
```

This prints out the following output:

```
Pyrrhic victory
```

Matches are even more powerful than this; the expression that is being matched can be destructured in the left-hand side, and this can even be combined with `if` conditions called **guards**:

```
 let loki = ("Loki", true, 800u32);
    match loki {
        (name, demi, _) if demi => {
                              print!("This is a demigod ");
                              println!("called {}", name);
                          },
        (name, _, _) if name == "Thor" =>
println!("This is Thor!"),
        (_, _, pow) if pow <= 1000 =>
```

```
println!("This is a powerless god"),
        _ => println!("This is something else")
    }
```

This prints out the following output:

```
    This is a demigod called Loki
```

Note that because `demi` is a Boolean we don't have to write `if demi == true`. If you want to do nothing in a branch, then write `=> {}`.

Destructuring works not only for tuples like in this example, but can also be applied for structs:

```
struct Point {
    x: i32,
    y: i32
}

let origin = Point { x: 0, y: 0 };
match origin {
Point { x: x, y: y } => println!("This is the point: ({},{})", x, y),
}
```

You can find a more elaborate example in `destructuring_structs.rs`.



Exercise:
What happens if you move the _ branch from the last position upwards?
See an example in `Chapter 4/exercises/pattern_match.rs`

Program arguments

Reading in program parameters from the command line at the startup of a program is easy in Rust, just use the method `std::env::args()`. We can use the function `collect()` to these parameters into a vector of `String`, like this:

```
// code from Chapter 4/code/arguments.rs:
use std::env;

fn main() {
   let args: Vec<String> = env::args().collect();
   println!("The program's name is: {}", args[0]);
   for arg in args.iter() {
         println!("Next argument is: {}", arg)
```

```
    }
    println!("Total arguments supplied: {}", args.len() - 1);
    for n in 1..args.len() {
       println!("The {}th argument is {}", n, args[n]);
    }
}
```

Call the program like this:

- `arguments arg1 arg2` on Windows
- `./arguments arg1 arg2` on Linux and OS X

Here is the output from a real call:

```
f:\>arguments Merlin Gandalf Sauron
The program's name is: arguments
Next argument is: arguments
Next argument is: Merlin
Next argument is: Gandalf
Next argument is: Sauron
I got 3 arguments:
The 1th argument is Merlin
The 2th argument is Gandalf
The 3th argument is Sauron
```

The argument `args[0]` is the program's name, the next arguments are the command-line parameters. We can iterate through the arguments or access them by index. The argument `args.len()- 1` ;gives us the number of parameters.

For more complex parsing with options and flags use the `getopts` or the `docopt` crate. To get started there is an example at `http://rustbyexample.com/arg/getopts.html`.

In the same way `env::vars()` returns the operating system environment variables:

```
let osvars = env::vars();
for (key, value) in osvars {
    println!("{}: {}", key, value);
}
```

This starts with printing out on Windows for example:

```
HOMEDRIVE: C:
USERNAME: CVO
LOGONSERVER: \\MicrosoftAccount
```

Summary

In this chapter, we increased our capabilities for working with composite data in Rust, from strings, arrays and vectors, and slices of both, to tuples, structs, and enums. We also discovered that pattern matching, combined with destructuring and guards, is a very powerful tool for writing clear and elegant code. We learned how to process input from the console with error-handling, and how to use program arguments.

In the following chapter, we will see that functions are much more powerful than we saw until now. Furthermore, we will discover that structs can have methods by implementing traits, almost like classes and interfaces in other languages.

5
Higher Order Functions and Error-Handling

Now that we have the data structures and control constructs in place, we can start discovering the functional features of Rust, which make it a really expressive language.

We will cover the following topics:

- Higher order functions and closures
- Iterators
- Consumers and adapters
- Generic data structures and functions
- Error-handling
- Some more examples of error-handling

Higher order functions and closures

By now, we know how to use functions, like in the following example where function `triples` changes our `strength` variable, but only if the return value of the function `triples` is assigned to the variable `strength`:

```
// see code in Chapter 5/code/higher_functions.rs
let mut strength = 26;
println!("My tripled strength equals {}",triples(strength)); // 78
println!("My strength is still {}", strength); // 26
strength = triples(strength);
println!("My strength is now {}", strength); // 78
```

Here, the function `triples` is defined as the following function:

```
fn triples(s: i32) -> i32 { 3 * s }
```

In the preceding code, `s` represents a value for the `strength` variable.

Suppose our player smashes an amazing power stone, so that his strength is tripled and the resulting strength tripled again; we could write it as follows:

```
triples(triples(s))
```

We can also write a function to do this, but it would be even more general to have a function, let's call it `again`, that could apply a certain function `f` of type `F` upon its result, enabling us to create all kinds of new game tricks, like this:

```
fn again (f: F, s: i32) -> i32  { f(f(s)) }
```

However, this is not enough information for Rust; the compiler asks us to say what that type `F` is. We can make this clear by adding `<F: Fn(i32) -> i32>` before the parameter list:

```
fn again<F: Fn(i32) -> i32>(f: F , s: i32) -> i32 {
    f(f(s))
}
```

The expression between `< >` (angle brackets) tells us that `F` is of type function `Fn` that takes an `i32` as parameter and returns an `i32` value.

Now, look at the definition of `triples`, that's exactly what this function does, the function `triples` has the signature of type `F`, so we can call again with `triples` as the first parameter:

```
strength = again(triples, strength);
println!("I got so lucky to turn my strength into {}", strength); // 702 (=
3 * 3 * 78)
```

The function `again` is an example of a **higher order function**, which is a function that takes another function (or more than one) as parameter.

Often, simple functions like `triples` are not even defined as a named function:

```
strength = 78;
let triples = |n| { 3 * n };
strength = again(triples, strength);
println!("My strength is now {}", strength); // 702
```

Here we have an anonymous function or closure `|n| { 3 * n }`, that takes a parameter `n` and returns its tripled value. The `||` (vertical bars) mark the start of a closure, and they contain the parameters passed to it (when there are no parameters it is written as `||`). There is no need to indicate the type of the parameters or that of the return value: a closure can infer these types from the context in which it is called.

The function `triples` is only a binding to a name, so that we can refer to the closure in other code. We could even leave that name out and inline the closure, like this:

```
strength = 78;
strength = again(|n| { 3 * n }, strength);
println!("My strength is now {}", strength); // 702
```

The closure is called with the parameter `n` taking the value of `s`, which is a copy of the variable `strength`. The braces can also be left out, simplifying the closure to the following:

```
strength = again(|n| 3 * n , strength);
```

Why is it called a closure? This becomes more apparent in the following example:

```
let x: i32 = 42;
let print_add = |s| {
    println!("x is {}", x);
    x + s
};
let res = print_add(strength); // <- here the closure is     //
assert_eq!(res, 744); // 42 + 702
```

This prints the following output:

```
x is 42
```

The closure `print_add()` has one argument and returns a 32 bit integer. The closure `print_add()` knows the value of `x` and all other variables that are available in its surrounding scope; it closes them in. A closure with no arguments has the empty parameter list `||`.

There is also a special kind of closure, called a moving closure, indicated by the `move` keyword. A normal closure only takes a reference to the variables it encloses, but a moving closure takes ownership of all the enclosing variables. It is self-contained and has its own stack-frame.

The preceding example would be written with a moving closure, like this:

```
let m: i32 = 42;
let print_add_move = move |s| {
    println!("m is {}", m);
    m + s
};
let res = print_add_move(strength); // strength == 702
m = 52;
assert_eq!(res, 744); // 42 + 702
```

Moving closures are mostly used when your program works with different concurrent threads (we will see more on this in `Chapter 9`, *Concurrency - Coding for Multicore Execution*).

As you will see in the following sections, higher order functions and closures are used throughout Rust, because they can make code much more concise and readable.

Iterators

An iterator is an object that returns the items of a collection in sequence, from the first to the last. To return the following item, it uses a `next()` method. Here, we have an opportunity to use an `Option`. Because an iterator can have no more values at the last `next()` call, `next()` always returns an `Option`, `Some(value)` when there is a value and `None` when there are no more values to return.

The simplest object that has this behavior is a range of numbers `0..n` (remember `n` is excluded). Every time we used a `for` loop like `for i in 0..n` the underlying iterator mechanism was put to work. Let's see an example:

```
// see code in Chapter 5/code/iterators.rs
let mut rng = 0..7;
println!("> {:?}", rng.next()); //
println!("> {:?}", rng.next()); //
for n in rng {
    print!("{} - ", n);
} // prints 2 - 3 - 4 - 5 - 6
```

This prints the following output:

```
Some(0)
Some(1)
```

We see here the function `next()` at work, producing `0` and `1`, and so on; the `for` loop continues until the end.



Exercise:
In the previous example we saw that the function `next()` returns a `Some` type value, a variant of type `Option` (see the *Result and Option* section in `Chapter 4`, *Structuring Data and Matching Patterns*). Write an endless loop over the range `rng` with the function `next()` and see what happens. How would you break the endless loop? Use a match on the `Option` value (for an example, see `Chapter 5/exercises/range_next.rs`). In fact, the `for` loop we saw right before this exercise is syntactic sugar for this `loop - match` construct.

Iterators are also the preferred way to loop over arrays or slices. Let's revisit the `aliens` array from `Chapter 4`, *Structuring Data and Matching Patterns*:

```
let aliens = ["Cherfer", "Fynock", "Shirack", "Zuxu"];"
```

Instead of using the index to show all the items one by one, let's do it the iterator way with the `iter()` function:

```
for alien in aliens.iter() {
   print!("{} / ", alien)
   // process alien
}
```

This prints out the following output:

```
Cherfer / Fynock / Shirack / Zuxu /
```

The `alien` variable is of type `&str`, a reference to each of the items in turn (technically, it is here of type `&str`, because the items themselves are of type `&str`, but this is not relevant to the point being made here). This is much more performant and safe than using an index, because now Rust doesn't need to do index-bounds checking, we're always certain to move within the memory of the array.

An even shorter way is to write the following:

```
for alien in &aliens {
    print!("{} / ", alien)
}
```

The variable `alien` is also of type `&str`, but the `print!` macro automatically dereferences this. If you want them to print out in reverse order, do `aliens.iter().rev()`. Other iterators we encountered in the previous chapter, where the `chars()` and `split()` methods on `Strings`.

Iterators are lazy by nature, they do not generate values unless asked and we ask by calling the `next()` method or applying a `for in` loop. That makes sense, we don't want to allocate one million integers in the following binding:

```
let rng = 0..1000_000; // _ makes the number 1000000 more readable
```

We want to allocate memory only when we need it.

Consumers and adapters

Now, we will see some examples that show why iterators are so useful. Iterators are lazy and have to be activated by invoking a consumer to start using the values. Let's start with a range of the numbers from `0` to `999` included. To make this into a vector, we apply the function `collect()` consumer:

```
// see code in Chapter 5/code/adapters_consumers.rs
let rng = 0..1000;
let rngvec: Vec<i32> = rng.collect();
// alternative notation:
// let rngvec = rng.collect::<Vec<i32>>();
println!("{:?}", rngvec);
```

This prints out the range (we shortened the output with...):

```
[0, 1, 2, 3, 4, ... , 999]
```

The function `collect()` loops through the entire iterator, collecting all of the elements into a container, here of type `Vec<i32>`. That container does not have to be an iterator. Notice that we indicate the item type of the vector with `Vec<i32>`, but we could have also written it as `Vec<_>`. The notation `collect::<Vec<i32>>()` is new, it indicates that the function `collect()` is a parametrized method that can work with generic types. See the next section for more information.

The function `find()` consumer gets the first value of the iterator that makes its condition (here `>= 42`) true and returns it as a value of the type `Option`, for example:

```
let forty_two = rng.find(|n| *n >= 42);
println!("{:?}", forty_two);  //
```

This prints out the following output:

```
Some(42)
```

The value of the function `find()` is of type `Option` because the condition could be false for all items and then it would return a `None` value. The condition is wrapped in a closure `|n| *n >= 42`, which is applied on every item of the iterator through a reference `n`; that's why we have to dereference `*n` to get the value.

Suppose we only want the even numbers in our range, producing a new range by testing a closure condition on each item. This can be done with the `filter()` function, which is an adapter because it produces a new iterator from the old one. Its result can be collected just like any iterator:

```
let rng_even = rng.filter(|n| is_even(*n))
                  .collect::<Vec<i32>>(); // (1)
println!("{:?}", rng_even);
```

Here, `is_even` is the following function:

```
fn is_even(n: i32) -> bool {
    n % 2 == 0
}
```

This prints out the following output:

```
[0, 2, 4, ..., 996, 998]
```

This shows that odd integers are filtered out.

Notice how we can chain our consumers and adapters, just apply `collect()` on the result of the `filter()` function with `.collect()`, as in the previous *line 1*.

What if we want to cube (`n * n * n`) every item in the resulting iterator? Producing a new range by applying a closure to each item in it can be done with the `map()` function:

```
let rng_even_pow3 = rng.filter(|n| is_even(*n))
                       .map(|n| n * n * n)
                       .collect::<Vec<i32>>();
println!("{:?}", rng_even_pow3);
```

This prints out the following output:

```
[0, 8, 64, ..., 988047936, 994011992]
```

If you only want the first five results, insert a `take(5)` adapter before the `collect`. The resulting vector then contains the following:

```
[0, 8, 64, 216, 512]
```

So, call a consumer if you see the following message while compiling:

```
warning: unused result which must be used: iterator adaptors are lazy and
do nothing unless consumed
```

To see all consumers and adapters, consult the docs of module `std::iter`.



Exercise:

Another very powerful consumer is the `fold()` function.
The following example calculates the sum of the first hundred integers. It starts with a base value 0, which is also the initial value of the accumulator `sum`, and then iterates and adds every item `n` to `sum`:

```
let sum = (0..101).fold(0, |sum, n| sum + n);
println!("{}", sum); //prints out 5050
```

Now calculate the product of all cubes of the integers in the range from 1 to 6.
The result should be `1728000`; watch out for the base value! As a second exercise, subtract all items from the following array `[1, 9, 2, 3, 14, 12]` starting from `0` (that is, `0 -1 -9 -2` and so on).
This should result in `41` (for example code, see `Chapter 5/exercises/fold.rs`).

Generic data structures and functions

Genericity is the capacity to write code once, with types not or partly specified, so that the code can be used for many different types. Rust has this capacity in abundance, applying it for both data structures and functions.

A composite data structure is generic if the type of its items can be of a general type `<T>`. The type `T` can for example be an `i32` value, an `f64`, a `String`, but also a `struct` type like `Person` that we have coded ourselves. So, we can have a vector `Vec<f64>`, but also a vector `Vec<Person>`. If you make `T` a concrete type, then you must substitute the type `T` with that type everywhere `T` appears in the definition of the data structure.

Our data structure can be parametrized with a generic type `<T>`, and so has multiple concrete definitions--it is polymorphic. Rust makes extensive use of this concept, which we encountered already in `Chapter 4`, *Structuring Data and Matching Patterns* when we talked about arrays, vectors, slices and the `Result` and `Option` types.

Suppose you want to define a `struct` with two fields, `first` and `second`, but you want to keep the type of these fields generic. We can define it as follows:

```
// see code in Chapter 5/code/generics.rs
struct Pair<T> {
    first: T,
    second: T,
}
```

We can now define a `Pair` of magic numbers, or a `Pair` of `magicians`, or whatever we want, like this:

```
let magic_pair: Pair<u32> = Pair { first: 7, second: 42 };
let pair_of_magicians: Pair<&str> = Pair { first: "Gandalf", second:
"Sauron" };
```

What if we wanted to write functions that work with generic data structures? They would also have to be generic, right? As a simple example, how would we write a function that returns the second item of a `Pair`? We can do it like this:

```
fn second<T>(pair: Pair<T>) {
  pair.second;
}
```

We could also call the `second` function as follows:

```
let a = second(magic_pair);
```

This produces the following output:

```
42
```



Note the type `<T>` right after the function name `second`; this is how generic functions are declared.

Let's now investigate why `Option` and `Result` types are so powerful. Here is the definition of the `Option` type again:

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

From this, we can define multiple concrete types, as follows:

```
let x: Option<i8> = Some(5);
let pi: Option<f64> = Some(3.14159265359);
let none: Option<f64> = None;
let none2 = None::<f64>;
let name: Option<&str> = Some("Joyce");
```

When the type does not correspond with the value, a mismatched types error occurs, as in the following code:

```
let magic: Option<f32> = Some(42)
```

We can define a `struct` `Person` as follows:

```
struct Person {
  name: &'static str,
  id:   i32
}
```

Then, we can add a few `Person` objects using `let` binding:

```
let p1 = Person{name: "James Bond", id: 7};
let p2 = Person{name: "Vin Diesel", id: 12};
let p3 = Person{name: "Robin Hood", id: 42};
```

Then, we can make an `Option` type value or a vector for the struct `Person` as the following:

```
let op1: Option<Person> = Some(p1);
let pvec: Vec<Person> = vec![p2, p3];
```

You would use the `Option` type in a situation where you expect to get a value, but when there is always a possibility that no value will be given. A typical scenario would be user input.

Error-handling

A Rust program must be maximally prepared to handle unforeseen errors, but unexpected things can always happen, like integer division by zero:

```
// see code in Chapter 5/code/error_div.rs
let x = 3;
let y = 0;
x / y;
```

The program stops with the following message:

```
thread '<main>' panicked at 'attempted to divide by zero'
```

Panics

A situation could occur that is so bad (like when dividing by zero) that it is no longer useful to continue running the program: we cannot recover from the error. In the case of such an error, we can invoke the `panic!("message")` macro, which will release all resources owned by the thread, report the message, and then make the program exit. We could improve the previous code, like this:

```
// see code in Chapter 5/code/errors.rs
if (y == 0) { panic!("Division by 0 occurred, exiting"); }
println!("{}", div(x, y));
```

The function `div` in the preceding code contains the following:

```
fn div(x: i32, y: i32) -> f32 {
    (x / y) as f32
}
```

A number of other macros, like the `assert!` family, can also be used to signal such unwanted conditions:

```
assert!(x == 5); //thread <main> panicked at assertion failed: x == 5
assert!( x == 5, "x is not equal to 5!");
// thread <main> panicked at "x is not equal to 5!"
assert_eq!(x, 5); // thread '<main>' panicked at 'assertion failed: (left:
`3`, right: `5`)',
```

When the condition is not true, they result in a panic situation and exit. The error message given as the second parameter of `assert!` will be printed out, if it is present. Otherwise, the general message `assertion failed` is given. The `assert!` macro is mostly useful to test for pre and post conditions in and around functions.

Portions of code that normally would not be executed can contain the `unreachable!` macro, which will panic when it is executed:

```
unreachable!();
// thread '<main>' panicked at 'internal error: entered         //
unreachable code'
```

Testing for failure

However, in most cases, we would like to attempt to recover from the error and let the program continue. We have already seen the basic techniques used in Rust to do just this in the *Result and Option* section in `Chapter 4`, *Structuring Data and Matching Patterns*, and in the section *Generic data structures and functions* in this chapter.

Both the `Option` and `Result` types are generic types:

- The `Option<T>` enum can be used when we expect a value, at which point a `Some(T)` value is given, and a `None` value is returned when there was no value or in the failure case. In this way Rust forces nothingness to appear in a clear and syntactically identifiable form, leaving no room for null pointer runtime errors.
- The `Result<T, E>` enum can be used to return an `Ok(T)` value in the normal (success) case, and an `Err(E)` value in the failure case, containing info on the error.

It is used when a computation should return a result, but it can also return an error if something went wrong. The `Result` type is defined with two generic types `T` and `E` as:

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

It again shows Rust's commitment to being on the safe side: if it's OK, give back a value of type `T`, if there is a problem, then give back the error which is a value of type `E` (usually an error message string). So, we could read them also as: `OK(what)` and `Err(why)`, where `what` has type `T` and `why` has type `E`.

In the following example, we use the `Result` type to safely read in a value from the keyboard:

```
let input_num: Result<u32, _> = buf.trim().parse();
```

In other languages, like Java or C#, parsing the input to a number could result in an exception (when the input contains non-numeric characters or when it is nothing or null), and you would have to use a resource heavy `try` or `catch` construct to deal with it.

In Rust, the result of the `parse()` function is a `Result` type, and we simply test the return value of `Result` with a `match` as in line (1) in the following code, which is a much simpler mechanism.

Here is the complete code:

```
// see code in Chapter 5/code/input_number.rs
use std::io;

fn main() {
  println!("Give an integer number bigger than 0:");
  let mut buf = String::new();
  io::stdin().read_line(&mut buf)
      .ok()
      .expect("Failed to read line");
  let input_num: Result<u32, _>  = buf.trim().parse();
  let res = match input_num {                              // (1)
        Ok(num) => num,
        Err(_) => 0
    };
    if res != 0 {
        println!("{}, that's a beautiful number", res);
    }
    else {
 println!("The input was not correct: {:?} ",  input_num);
    }
}
```

We captured the result variable `num` in a new variable, using a `let` binding on the match:

```
let res = match input_num {
      Ok(num) => num,
      Err(_) => 0
};
```

This prints out the input number or an error message if the input is incorrect.

Some more examples of error-handling

Another way to obtain the same result is to wrap the dangerous read operation in a separate function, here the function `read_u32()` returns an `Option` type value to be tested in the main code:

```
// see code in Chapter 5/code/input_number2.rs
use std::io;
fn main() {
    println!("Give an integer number bigger than 0:");
    let num = read_u32();
    match num {
        Some(val) => println!("That's the number: {}", val),
```

```
        None => println!("Failed to read number.")
    }
}

fn read_u32() -> Option<u32> {
    let mut buf = String::new();
    if io::stdin().read_line(&mut buf).is_ok() {
        let result = buf.trim().parse::<u32>();
        return match result {                              // (1)
            Ok(value) => Some(value),
            Err(_) => None  //Failed to parse
        };
    }
    None //Failed to read from stream
}
```

In this code, we effectively transform a `Result` type value into an `Option` type value in the `match` as it can be seen in line (1).

Here is another example how we can use `Result` type to return an error condition and making a safe function for calculating the square root of a floating point number using the `std::f32::sqrt()` function:

```
// see code in Chapter 5/code/sqrt_match.rs
fn sqroot(r: f32) -> Result&lt;f32, String> {
if r < 0.0 {
return Err("Number cannot be negative!".to_string());
 }
    Ok(f32::sqrt(r))
}
```

We guard against taking the square root of a negative number (which would give `NAN`--Not a Number) by returning an `Err` value.

```
let m = sqroot(42.0);
```

In the calling code, we use our trusted pattern match mechanism to distinguish between the two cases:

```
match m {
   Ok(sq) => println!("The square root of 42 is {}", sq),
   Err(str) => println!("{}", str)
}
```

So, in the normal case, we get output as follows:

```
    The square root of 42 is 6.480741
```

With `let m = sqroot(-5.0);` the error message is printed as follows:

```
Number cannot be negative!
```



Why are `Option` and `Result` killer features of Rust? The use of `match` for both `Option` and `Result` type values ensures that no null values or errors can propagate through your code, leaving no room for null pointer runtime errors or other exceptions to crash your program.

Moving from a *crash on error* strategy to a strategy of actually *handling the error* means switching from `ok().expect()` or `ok().unwrap()` to a `match` statement that tests on the `Ok(T)` or `Err(E)` value outcome of the `Result` type value. You can still use the function `unwrap()` for quick prototyping.

Another way of handling a `Result` type value which shows more clearly why an errors occurs is as follows:

```
let result = match process_result_value {
      Err(why) => panic!("some error occurred: {}",
      Error::description(&why)),
      Ok(result) => result
};
```

If the program could, in some way, recover from the error, we could decide not to stop it here, but just return to the calling function, like this:

```
let result = match process_result_value() {
    Err(why) => { println!("some error occurred: {}",
                                Error::description(&why));
             return;
 },
    Ok(result) => result
};
```

In many cases, you can also just test the outcome of the operation with the following:

```
if process_result_value().is_err() {
  // process error value
}
```

This will return `false` in case of an error.

So, in Rust, errors are not exception objects like in other languages: that would not be possible because Rust has no virtual machine to catch exceptions. Rust uses a type-based approach to error-handling, instead of an exception being raised, an error is returned from a function in the `Result` type value, so that the developer can handle the error case.

Exceptions result in complicated control-flow and they don't work well with multithreaded code, Rust avoids this with its error-handling mechanism.

The try! macro and the ? operator

When, during a computation, several function calls return a `Result` type value, the handling of the error propagation through pattern matching can become quite tedious-each returned `Result` type value requires a match to differentiate between an `Ok` value and an `Err` value.

We had exactly such a situation in the `input_number.rs program`, where we read an input from the terminal and tried to parse it into an unsigned integer, both operations can fail.

The `try!` macro was specifically made in order to simplify such code and make error-handling more elegant and readable.

The `try!` macro can only be used from inside a function that returns a `Result` type value, so not from within the `main()` function. Also, it can only be used on functions that return a `Result` type value. It returns the `Ok` value, and then continues with the function's code. But in case of an error the `Err` value is returned, returning immediately from the enclosing function. If we apply this to `input_number.rs`, we get much more readable code, as follows:

```
// see code in Chapter 5/code/try_input_number.rs
use std::io;
use std::error;

fn main() {
    println!("Give a positive secret number: ");
    match input_num() {
        Ok(v) => println!("Input value is: {}", v),
  Err(e) => println!("Error - Please input an integer number!: {}", e)
    }
}

fn input_num() -> Result<u32, Box<error::Error>> {
    let mut input = String::new();
    try!(io::stdin().read_line(&mut input));
    Ok(try!(input.trim().parse()))
}
```

When there is no input, the program prints out the following error:

```
    Error - Please input an integer number!: cannot parse integer from
empty string
```

Then, the program exits normally. When a non-digit input is given, the program prints out to give a positive secret number, as follows:

```
    Error - Please input an integer number!: invalid digit found in string
```

Then, it exits normally.

The dangerous operations are wrapped inside the function `input_num`. We can use the `try!` macro on these operations because the `input_num` function returns a `Result` type value.

Our code becomes even shorter when we use the error propagation operator `?`, which works the same way as the `try!` macro:

```
  fn input_num() -> Result<u32, Box<error::Error>> {
       let mut input = String::new();
    io::stdin().read_line(&mut input)?;
    Ok(input.trim().parse()?)
}
```

The `Box<error::Error>>` is a so called boxed pointer, a reference to an `Error` instance from the `std::error` module. We talk more about the `Box` pointer in `Chapter 7`, *Ensuring Memory Safety and Pointers*. We need to use a `Box` pointer here to let the compiler know that we don't know the size of the error instance.

If all you need from errors is their string content, you could simplify error-handling by using the simple error crate (`https://crates.io/crates/simple-error`).

Summary

In this chapter, we learned all kinds of techniques to make our code more flexible, using higher-order functions, closures, iterators, and generic types and functions. We then reviewed the basic error-handling mechanisms, which make good use of generic types.

In the following chapter, we will expose the object-oriented nature of Rust, by defining methods on structs and implementing traits.

6
Using Traits and OOP in Rust

In this chapter, we explore the object-oriented features of Rust, which make it a really expressive language. With these features, we can apply well-known techniques from the object-oriented world, so that our programs can better model real-world situations.

We will cover the following topics:

- Associated functions on structs
- Methods on structs
- Using a constructor pattern
- Using a builder pattern
- Methods on tuples and enums
- Traits
- Using trait constraints
- Static and dynamic dispatch
- Built-in traits and operator overloading
- OOP in Rust
- Inheritance with traits
- Using the visitor pattern

Associated functions on structs

Rust makes it possible to call a method in two ways. For example, when we want to obtain the length of a string, you can do:

```
// see code in Chapter 6/code/paradigm.rs
let str1 = "abc";
println!("{}", str::len(str1)); // 3
println!("{}", str1.len());     // 3
```

The first way is procedural and calls the `len` function from the `str` crate in the standard library and passes the string slice `str1` as a parameter. The second way which is more object-oriented and more commonly used calls the `len` method on the string slice `str1`. If you look it up in the API docs, you can see it has the signature:

```
    fn len(&self) -> usize
```

It effectively takes a reference (`&`) to `self` as a parameter.

So we see that Rust caters also for more object-oriented developers, who are used to the `object.method()` type of notation instead of `function(object)` type of notation. In Rust, we can define associated functions and methods on a `struct`, which pretty much compares to the traditional class and methods concept.

Suppose we are building a game that takes place on a planet in a distant solar system, inhabited by hostile aliens. Let's define a `struct Alien` like this:

```
// see code in Chapter 6/code/methods.rs
struct Alien {
   health: u32,
   damage: u32
}
```

Where the variable `health` is the `Alien` struct's condition, and the variable `damage` is the amount your health is decreased when the alien attacks. We could make an `Alien` like this:

```
let mut bork = Alien{ health: 100, damage: 5 };
```

The variable `health` cannot be more than `100`, but we cannot impose this constraint when making a `struct` instance like this.

A solution is to define a new function for the `Alien` implementor, where we can test the value before it is being set:

```
impl Alien {
   fn new(mut h: u32, d: u32) -> Alien {
      // constraints:
      if h > 100 { h = 100; }
      Alien {health: h, damage: d}
   }
}
```

And then construct a new `Alien` as follows:

```
let mut berserk = Alien::new(150, 15);
```

We define the new function (and all other functions and methods on the implementor `Alien`) inside an `impl Alien` block, which is separate from the `Alien` struct definition. It returns an `Alien` object after all constraints have been applied. We call it on the `Alien` struct itself as `Alien::new()`. It is a static function because we don't call it on an `Alien` strcuct instance.

If we look at our new `Alien` struct, we see that the constraint was imposed:

```
println!("The berserk's health at birth is: {}", berserk.health);
```

This has the output as follows:

```
The berserk's health at birth is: 100.
```

Such a new function closely resembles a constructor from object-oriented languages. The fact that it is called new is merely by convention, we could have called it as the `create()`, or the `give_birth()` function.

Another static function could be a warning given by all aliens:

```
fn warn() -> &'static str {
    "Leave this planet immediately or perish!"
}
```

This can be called like this:

```
println!("{}", Alien::warn());
```

And this outputs the message:

```
Leave this planet immediately or perish!
```

Methods on structs

All the functions defined for our `struct` until now are so called **associated functions**, they are associated with the `struct` and are called with the syntax:

```
Struct_name::ass_function()
```

We can also define real methods in Rust, that are called on a `struct` instance and that have a reference to that instance `&self` as first parameter.

When a specific struct `Alien` attacks, we can define a method for that `Alien` struct like this:

```
fn attack(&self) {
    println!("I attack! Your health lowers with {} damage points.",
self.damage);
}
```

And call it on the function `alien berserk` as follows:

```
berserk.attack();
```

A reference to `berserk` object (the `Alien` object on which the method is invoked) is passed as `&self` to the method. In fact the `self` object is like the `self` object in Python or `this` in Java or C#. A method always has `&self` instance as a parameter, in contrast to a static method.

Here the object is passed immutably, but what if attacking also lowers the `Alien` structs own `health`? Let's add a second `attack` method:

```
fn attack(&self) {
    self.health -= 10;
}
```

But Rust rejects this with two compiler errors. First it says the following:

```
    cannot assign to immutable field self.health
```

This we can remedy by passing a mutable reference to the `self` instance like this:

```
fn attack(&mut self)
```

But now Rust complains:

```
    duplicate definition of value 'attack'
```

This means that Rust does not allow two methods with the same name, as there is no method overloading in Rust. This is because of the way type inference works.

Changing the name to `attack_and_suffer`, we get:

```
fn attack_and_suffer(&mut self, damage_from_other: u32) {
   println!("I attack! Your health lowers with {} damage points.",
self.damage);
   self.health -= damage_from_other;
   }
```

After calling the `berserk.attack_and_suffer(31);` function; the `berserk` variables's `health` is now `69` (where `31` is the number of damage points inflicted upon the object `berserk` by another attacking `Alien`).



Exercise:

Complex numbers like 2 + 5i (i is the square root of -1) have a real part (here 2) and an imaginary part (5); both are floating point numbers. Define a `struct Complex` and some methods for it, for example: A new method to construct a complex number a `to_string` method that prints a complex number like 2 + 5i or 2 - 5i (hint: use the `format!` macro which works the same way as the `println!` macro but returns a string). An `add` method to add two complex numbers; this is a new complex number where the real part is the sum of the real parts of the operands, and the same for the imaginary part. A `times_ten` method that changes the object itself by multiplying both parts by 10 (hint: think carefully about the method's argument). As a bonus, make an `abs` method that calculates the absolute value of a complex number (see: `http://en.wikipedia.org/wiki/Absolute_value`) Note: If you need to work with complex numbers in a real project use the `Complex` struct from the `num` crate `num::Complex` (see `https://rust-num.github.io/num/num/struct.Complex.html`). Test your methods! (for example code see `Chapter 6/exercises/complex.rs`).

Using a constructor pattern

No method overloading means that we can only define one new function (which is optional anyway). We could invent different names for our constructors, which is good from the point of view of code documentation. This is demonstrated in the following example:

```
// see code in Chapter 6/code/constructor_pattern.rs
struct Alien {
    name: &'static str,
    health: u32,
```

```
    damage: u32
}

impl Alien {
    fn new(s: &'static str, mut h: u32, d: u32) -> Self {
        if h > 100 { h = 100; }
        Alien { name: s, health: h, damage: d }
    }

    pub fn default() -> Self {
        Alien::new("Walker", 100, 10)
    }
    pub fn give_health(h: u32) -> Self {
        Alien::new("Zombie", h, 5)
    }
}

fn main() {
    let al1 = Alien{ name: "Bork", health: 100, damage: 5 };
    let al2 = Alien::new("Berserk", 150, 15);
    println!("Alien 1 is a {} and inflicts {} damage points", al1.name,
al1.damage);
    let al3 = Alien::default();
    println!("Alien 3 is a {} and inflicts {} damage points", al3.name,
al3.damage);
    let al4 = Alien::give_health(75);
    println!("Alien 4 is a {} and inflicts {} damage points", al4.name,
al4.damage);
 }
```

This prints out the following output:

```
Alien 1 is a Bork and inflicts 5 damage points
Alien 3 is a Walker and inflicts 10 damage points
Alien 4 is a Zombie and inflicts 5 damage points
```

Here our new constructor is not public, so an `Alien` implementor cannot be constructed in code that uses our code. But the `default()` and `give_health()` methods are public (indicated by the keyword `pub`, as in `pub fn default()`), so external code is restricted in creating either a default Walker or a Zombie with a given health amount.

We see that all three constructors return a `Self` type, which is the `Alien` struct type in our context.

Another way to go is called the **builder pattern**, which we discuss in the following section.

Using a builder pattern

Sometimes data structures have lots of or complicated fields, so that they need a number of constructors to effectively create them. Other languages would use method overloading or named arguments, but Rust doesn't have these. Instead, we can use the so-called *builder pattern*, which is used occasionally in Servo and the Rust standard library. The idea is that the instance construction is forwarded to an analogous Builder struct, which has a default new constructor, methods that change each of its properties, and a finish method that returns an instance of the original struct. Here is an example:

```
// see code in Chapter 6/code/builder_pattern.rs
struct Alien {
    name: &'static str,
    health: u32,
    damage: u32
}

struct AlienBuilder {
    name: &'static str,
    health: u32,
    damage: u32
}

impl AlienBuilder {
    fn new() -> Self {
        AlienBuilder { name: "Walker", health: 100, damage: 10 }
    }

    fn name(&mut self, n: &'static str) -> &mut AlienBuilder {
        self.name = n;
        self
    }

    fn health(&mut self, h: u32) -> &mut AlienBuilder {
        self.health = h;
        self
    }

    fn damage(&mut self, d: u32) -> &mut AlienBuilder {
        self.damage = d;
        self
    }

    fn finish(&self) -> Alien {
        Alien { name: self.name, health: self.health, damage: self.damage }
    }
}
```

```
fn main() {
    let al1 = AlienBuilder::new()
                .name("Bork")
                .health(80)
                .damage(20)
                .finish();
    println!("name: {}", al1.name);
    println!("health: {}", al1.health);
}
```

This prints out the following output:

```
name: Bork
health: 80
```

This way we have used the type system to enforce our concerns, and we can use the methods on the struct `AlienBuilder` to constrain making the struct `Alien` instances in any way we choose.

Methods on tuples and enums

In Rust, methods cannot only be defined on structs, they can also be defined on tuples and enums, and even on built-in types like integers.

Here is an example of an instance method `mood` defined on the variants of an `Day` enum. It matches the variant to print out the `mood` string of the day:

```
// see code in Chapter 6/code/method_enum.rs
enum Day {
    Monday,
    Tuesday,
    Wednesday,
    Thursday,
    Friday,
    Saturday,
    Sunday,
}

impl Day {
    fn mood(&self) {
        println!("{}", match *self {
            Day::Friday => "it's friday!",
            Day::Saturday | Day::Sunday => "weekend :-)",
            _ => "weekday...",
        })
```

```
        }
    }

    fn main() {
        let today = Day::Tuesday;
        today.mood();
    }
```

This prints out the following output:

```
    weekday...
```

Traits

What if our game is really diversely populated, and besides Aliens we have also Zombies and Predators, and, needless to say, they all want to attack. Can we abstract their common behavior into something they all share? Of course, in Rust we say that they have a **trait** in common, analogous to an interface or superclass in other languages. Let's call that trait `Monster`, and because they all want to attack, a first version could be:

```
// see code in Chapter 6/code/traits.rs
trait Monster {
    fn attack(&self);
}
```

A trait mostly contains a description of methods, that is, their type declarations or signatures, but no real implementation (as we will see later in the example, a trait can contain a default implementation of a method). This is logical, because `Zombies`, `Predators`, and `Aliens` could each have their own method of attack. So there is no code body between `{}` after the function signature, but don't forget the `;` (semicolon) to close it off.

When we want to implement the `Monster` trait for the struct `Alien`, we write the following code:

```
impl Monster for Alien {

}
```

When we compile this, Rust throws the following error:

```
    not all trait items implemented, missing: `attack`
```

That's useful because Rust reminds us which methods from a trait we have forgotten to implement. The following code would make it pass:

```
impl Monster for Alien {
   fn attack(&self) {
println!("I attack! Your health lowers with {} damage points.",
self.damage);
   }
}
```

So the trait implementation for a type must provide the real code, which is executed when that method is called on an `Alien` struct object. If a `Zombie` struct attack is twice as bad, its `Monster` implementation could be:

```
impl Monster for Zombie {
   fn attack(&self) {
println!("I bite you! Your health lowers with {} damage points.", 2 *
self.damage);
   }
}
```

We could add other functions and methods to our trait, such as a `new`, a `noise`, and an `attack_with_sound` method:

```
trait Monster {
    fn new(hlt: u32, dam: u32) -> Self;
    fn attack(&self);
    fn noise(&self) -> &'static str;
    fn attacks_with_sound(&self) {
println!("The Monster attacks by making an awkward sound {}",
self.noise());
    }
}
```

Notice that in the `new` method the resulting object is of `Self` type, which becomes the `Alien` implementor type or `Zombie` in a real implementation of the trait.

Methods differ from functions because they have an `&self` instance as parameter, that means they have the object on which they are invoked as parameter, for example:

```
fn noise(&self) -> &'static str
```

When we call it with the `zmb1.noise()` function, the object `zmb1` becomes a `self` type.

A trait can provide default code for a method (like here for the `attack_with_sound` method). The implementor type can choose to take this default code or override it with its own version. Code in a trait method can also call upon other methods in the trait with the method `self.method()`, as in the `attack_with_sound` method where the `self.noise()` method is called.

The full implementation of the `Monster` trait for type `Zombie` could then be:

```
impl Monster for Zombie {
   fn new(mut h: u32, d: u32) -> Zombie {
      // constraints:
      if h > 100 { h = 100; }
      Zombie {health: h, damage: d}
   }

   fn attack(&self) {
println!("The Zombie bites! Your health lowers with {} damage points.", 2 *
self.damage);
   }

   fn noise(&self) -> &'static str {
      "Aaargh!"
   }
}
```

And here is a short fragment of our game scenario:

```
let zmb1 = Zombie {health: 75, damage: 15};
println!("Oh no, I hear: {}", zmb1.noise());
zmb1.attack();
```

This prints the following output:

```
Oh no, I hear: Aaargh!
The Zombie bites! Your health lowers with 30 damage points.
```

Traits are not limited to structs, they can be implemented on any type. A type can also implement many different traits. All the different implemented methods are compiled to a version specific for their type, so after compilation there exists a different `new` method for the `Alien`, `Zombie`, and `Predator` trait.

Implementing all of the methods in a trait can be tedious work. For example, we probably want to be able to show our creatures in this way:

```
println!("{:?}", zmb1);
```

Unfortunately this gives us the compiler error:

```
    error[E0277]: the trait bound `Zombie: std::fmt::Debug` is not
satisfied.
```

So from the message we can infer that this `{:?}` format string uses a trait `Debug`. If we look this up in the docs we find that we must implement an `fmt` method (specifying a way to format the data from the struct instance). But the compiler once again helps us out here: if we prefix our `Zombie` struct definition with `#[derive(Debug)]`, then a default code version is generated automatically!

```
#[derive(Debug)]
struct Zombie { health: u32, damage: u32 }
```

So that `println!("{:?}", zmb1);` now shows:

```
    Zombie { health: 75, damage: 15 }
```

This mechanism also works for a whole list of other traits (see section *Built-in traits* and `http://rustbyexample.com/trait/derive.html`).

The actual implementation of a trait for a type is separated from the definition of the type itself: the data and the operations on them are independent. They can reside in different files, or even different modules, providing for more flexibility.

Using trait constraints

Back in the section on *Generic data structures and functions* in `Chapter 5`, *Higher Order Functions and Error-Handling*, we made a function `sqroot` to calculate the square root of a 32-bit floating point number:

```
// see code in Chapter 5/code/sqrt_match.rs
Use std::f32;

fn sqroot(r: f32) -> Result<f32, String> {
if r < 0.0 {
return Err("Number cannot be negative!".to_string());
}
   Ok(f32::sqrt(r))
}
```

What if we wanted to calculate the square root of an `f64` type number? It would be very unpractical to make a different version of the function for each type. A first attempt would be to just replace an `f32` type with a generic type `<T>`:

```
// see code in Chapter 6/code/trait_constraints.rsfn sqroot<T>(r: T) ->
Result<T, String> {
   if r < 0.0 {
      return Err("Number cannot be negative!".to_string());
   }
    Ok(T::sqrt(r))
}
```

But Rust does not agree because it doesn't know anything about `T`, signaling multiple errors:

```
      binary operation `<` cannot be applied to type `T`
    and no function or associated item named `sqrt` found for type `T` in
    the current scope
    ...
```

A `Float` trait exists that would be general enough, but not in the standard library, it lives in the `num` crate, more specifically as `num::traits::Float`. To be able to use that crate, we must create a project with `cargo`:

```
cargo new trait_constraints – –bin
```

Then we have to edit the `Cargo.toml` file and add the following section:

```
[dependencies]
num = "*"
```

Now give the commands `cargo update` to add crate `num` to your project.

To be able to use our external crate `num`, we add the following code at the start of our file:

```
extern crate num;
use num::traits::Float;
```

We can assert that `T` must implement this trait as follows:

```
fn sqroot<T: num::traits::Float>(r: T) -> Result<T, String> {...}
```

This is called putting a **trait constraint** or a **trait bound** on the type `T`. We assert that type `T` implements the trait `num::traits::Float`, and it ensures that the function can use all methods of the specified trait.

To be as general as possible, we also use the special indicator for 0 that exists in the `num` crate, named `num::zero()`, so our function now becomes:

```
fn sqroot<T: num::traits::Float>(r: T) -> Result<T, String> {
      if r < num::zero() {
            return Err("Number cannot be negative!".to_string());
      }
      Ok(num::traits::Float::sqrt(r))
}
```

This works for both of the following calls:

```
println!("The square root of {} is {:?}", 42.0f32, sqroot(42.0f32) );
println!("The square root of {} is {:?}", 42.0f64, sqroot(42.0f64) );
```

This produces the following output:

```
The square root of 42 is Ok(6.480741)
The square root of 42 is Ok(6.48074069840786)
```

But if we try to call the function `sqroot` on an integer, like this:

```
println!("The square root of {} is {:?}", 42, sqroot(42) );
```

We get the following error:

```
the trait `num::Float` is not implemented for {integer}
```

Because an integer is not a `Float` type.

Another way to write the same trait constraint is with a `where` clause like this:

```
fn sqroot<T>(r: T) -> Result<T, String> where T: Float { ... }
```

Why does this other form exist? Well, there can be more than one generic type `T` and `U`, and each type can be constrained to multiple traits (indicated by a `+` between the traits) `Trait1`, `Trait2`, and so on, like in this fictitious example:

```
fn multc<T: Trait1, U: Trait1 + Trait2>(x: T, y: U) {...}
```

With the `where` syntax this can be made much more readable, like this:

```
fn multc<T, U>(x: T, y: U) where T: Trait1, U: Trait1 + Trait2 {...}
```



Exercise:

Define a trait `Draw` with a `draw` method.
Define struct types `S1` with an integer field, and `S2` with a float field.
Implement the trait `Draw` for `S1` and `S2` (`draw` prints the values out surrounded by `***`).
Make a generic `draw_object` function that takes any object that implements `Draw`.
Test it out! (see example code in `Chapter 6/exercises/draw_trait.rs`)

Static and dynamic dispatch

Our function `sqroot` from the previous section is generic and works for any `Float` type. The compiler creates a different executable `sqroot` method for any type it is supposed to work with, in this case the `f32` and `f64` type. Rust applies this mechanism when a function call is **polymorphic**, that is when a function can accept arguments of different type. This is called static dispatch (also called compile-time polymorphism) and there is no runtime overhead involved. This is in contrast to how Java interfaces work, where the dispatching is done dynamically in runtime by the JVM. However Rust also has a form of dynamic dispatch (also called runtime polymorphism), using so called **trait objects**.

For an example of static and dynamic dispatch, see the following code snippet:

```
// see code in Chapter 6/code/dispatch.rs
struct Circle;
struct Triangle;

trait Figure {
    fn print(&self);
}

impl Figure for Circle {
    fn print(&self) {
        println!("Circle");
    }
}

impl Figure for Triangle {
    fn print(&self) {
        println!("Triangle");
    }
}
```

```
// static dispatch with trait bounds
fn log<T: Figure>(figure: &T) {
    figure.print();
}

// dynamic dispatch: function takes a trait object
fn logd(figure: &Figure) {
    figure.print();
}

fn main() {
// static dispatch
    let circle   = Circle;
    let triangle = Triangle;
    log(&circle);
    log(&triangle);

// dynamic dispatch:
    let mut figures: Vec*Box*Figure** = Vec::new();
    figures.push(Box::new(Circle));
    figures.push(Box::new(Triangle));
    // the precise type of figure can only be known at runtime:
    for figure in figures {
        logd(&*figure);
    }
}
```

This produces the following output:

```
Circle
Triangle
Circle
Triangle
```

With static dispatch, there is a compiled version for each specific type for which the function is called. These calls can be inlined, which guarantees the best performance. On the other hand the binary code size is increased. Dynamic dispatch is used when the precise type on which a function is called can only be known at runtime. In our example the trait object is a reference to an instance implementing the `Figure` trait. Because calls are resolved at runtime, dynamic dispatch is not so performant as its static variant, but sometimes there is no other option.



More detailed info can be found at `https://doc.rust-lang.org/1.0.0-beta/book/static-and-dynamic-dispatch.html`.

Built-in traits and operator overloading

The Rust standard library is packed with traits, which are used all over the place. For example, there are traits for which the compiler is capable of providing a basic implementation with a `#[derive]` attribute, as we saw in the section on *Traits*:

- **Comparing instances**: The `Eq` and `PartialEq` trait
- **Ordering instances**: The `Ord` and `PartialOrd` trait
- **Creating an empty instance**: The `Default` trait
- **To create a zero instance of a numeric data type**: The `Zero` trait

The next chapter shows an example of how to implement the following three traits:

- **Formatting a value using {:?}**: The `Debug` trait, defining an `fmt` method
- **Copy an instance**: The `Copy` trait
- **Create a duplicate instance**: The `Clone` trait
- **Computing a hash**: The `Hash` trait
- **Adding instances**: The `Add` trait, defining an `add` method. The `+` operator is just a nice way to use `add: n + m` is the same as `n.add(m)`. So if we implement the `Add` trait, we can use the `+` operator, this is called **operator overloading**.
    - The `Add` trait has the following signature:

```
pub trait Add<RHS = Self> {
     type Output;
      fn add(self, rhs: RHS) -> Self::Output;
}
```

So the `add` method and the trait item `Output` must both be implemented. The code `impl_add.rs` shows an implementation of the `Add` trait:

```
struct AType {
    value: i32,
}

impl AType {
    fn new(value: i32) -> AType {
        AType { value: value }
    }
}

impl Add for AType {
    type Output = AType;
```

```
    fn add(self, other: AType) -> AType {
        AType { value: self.value + other.value }
    }
}

fn main() {
    let at1 = AType{ value: 7 };
    let at2 = AType{ value: 42 };
    let at3 = at1.add(at2);
    println!("{:?}", at3);
    let at4 = AType{ value: 2 };
    let at5 = AType{ value: 108 };
    let at6 = at4 + at5;
    println!("{:?}", at6);
}
```

- This prints the following output:

```
AType { value: 49 }
AType { value: 110 }
```



Note that we can use it as a `+` operator or an `add` method. A lot of other traits can also be used to overload operators, such as `Sub(-)`, `Mul(*)`, `Deref (*v)`, `Index([])`, and so on.

- **Freeing resources of an object (when it goes out of scope)**: The `Drop` trait, in other words the object has a destructor. Freeing resources is taken care of automatically by the compiler, but by implementing `Drop` you can do additional things.

In the section *Iterators* we described how an iterator works and used it on ranges and arrays. In fact an iterator is also defined as a trait in Rust in `std::iter::Iterator`. From the docs for iterator (see `http://doc.rust-lang.org/std/iter/trait.Iterator.html`) we see that we only need to define the `next()` method, which advances the iterator to return the next value as an `Option` type. When the `next()` method is implemented for the type of your object, we can then use a `for in` loop to iterate over the object.

OOP in Rust

While there is no unique definition of what object-orientation in a programming language is, it is clear by now that in Rust you can express several important OO-concepts.

- Objects with data (their state) and methods (their behavior):
    - Rust has these--structs (and other types such as enums) have data and `impl` blocks provide methods on them. The definitions of structure and behavior are separate.
- Encapsulation of data and implementation:
    - In other words--external code can only change or interact with an object through its public methods. In Rust by default everything is private, so not accessible from the outside. You can decide which types, functions, methods and modules are public by adding a `pub` keyword before their definition. For more detail, see `Chapter 8`, *Organizing Code and Macros* in the section, *Visibility of items*.
- Inheritance to promote code reuse and use polymorphism:
    - In Rust, inheritance, strictly speaking, does not exist: a struct cannot inherit its fields or methods from another struct. But the implementation of traits by structs can almost provide the same benefits. Indeed traits can inherit from other traits; see the next section *Inheritance with traits*.

Code reuse among structs can be obtained by letting them implement the same trait(s), because then they share the methods with default code from these trait(s). Polymorphism means objects of a different type can be treated as of one type, because they inherit from the same supertype or implement the same interface. As we saw in the section on *Static and Dynamic Dispatch*, Rust has trait objects to work with values of any type, as long as these values implement a particular trait.

We can conclude that, although the concept of a class and inheritance between classes is not present in Rust, nevertheless Rust enables the most important object-oriented concepts, more than in many other languages.

Inheritance with traits

Traits can also *inherit* from other traits, indicated with the `:` operator:

```
trait Trait1 : SuperTrait
```

Look at the following code, where `trait Monster` inherits from `trait SuperMonster`. In such a case any type that implements `Monster` must also implement `SuperMonster`, in this specific case its `super1()` method:

```
// see code in Chapter 6/code/super_traits.rs
struct Zombie { health: u32, damage: u32 }

trait SuperMonster {
    fn super1(&self);
}

trait Monster : SuperMonster {
    fn new(hlt: u32, dam: u32) -> Self;
    fn attack(&self);
    fn noise(&self) -> &'static str;
}

impl SuperMonster for Zombie {
    fn super1(&self) {
        println!("I am a SuperMonster");
    }
}

impl Monster for Zombie {
   fn new(mut h: u32, d: u32) -> Zombie {
      if h > 100 { h = 100; }
      Zombie { health: h, damage: d }
   }

   fn attack(&self) {
      println!("The Zombie bites! Your health lowers with {} damage
points.", 2 * self.damage);
   }

   fn noise(&self) -> &'static str {
      "Aaargh!"
   }
}

fn main() {
   let zmb1 = Zombie { health: 75, damage: 15 };
   println!("Oh no, I hear: {}", zmb1.noise());
    zmb1.super1();
}
// Oh no, I hear: Aaargh!
// I am a SuperMonster
```

Using the visitor pattern

The visitor design pattern is a common pattern used in object-oriented languages. A visitor stores an algorithm that operates over a collection of objects of different types. It allows multiple different algorithms to be written over the same data without having to modify the code of the data's classes (For a more detailed description, see `https://en.wikipedia.org/wiki/Visitor_pattern`).

Let's implement a visitor to calculate the area of different geometric objects, such as squares and circles.

```
// see code in Chapter 6/code/visitor_pattern.rs
// structs:
struct Point {
    x: f64,
    y: f64
}

struct Circle {
    center: Point,
    radius: f64,
}

struct Square {
    lowerLeftCorner: Point,
    side: f64,
}

// traits:
trait ShapeVisitor {
    fn visit_circle(&mut self, c: &Circle);
    fn visit_square(&mut self, r: &Square);
}

trait Shape {
    fn accept<V: ShapeVisitor>(&self, sv: &mut V);
}

impl Shape for Circle {
    fn accept<V: ShapeVisitor>(&self, sv: &mut V) {
        sv.visit_circle(self);
    }
}

impl Shape for Square {
    fn accept<V: ShapeVisitor>(&self, sv: &mut V) {
        sv.visit_square(self);
```

```
    }
}

fn compute_area<S: Shape>(s: &S) -> f64 {
    struct AreaCalculator {
        area: f64,
    }

    impl ShapeVisitor for AreaCalculator {
        fn visit_circle(&mut self, c: &Circle) {
self.area = std::f64::consts::PI * c.radius * c.radius;
        }
        fn visit_square(&mut self, r: &Square) {
            self.area = r.side * r.side;
        }
    }
    let mut ac = AreaCalculator { area: 0.0 };
    s.accept(&mut ac);
    ac.area
}

fn main() {
    let cn = Point{ x: 0.0, y: 0.0 };
    let ci = Circle{ center: cn, radius: 1.0 };
    let area = compute_area(&ci);
    println!("The area of the circle is {}", area);
    let cn = Point{ x: 0.0, y: 0.0 };
    let sq = Square{ lowerLeftCorner: cn, side: 1.0 };
    let area = compute_area(&sq);
    println!("The area of the square is {}", area);

}
```

Which prints out the following output:

```
The area of the circle is 3.141592653589793
The area of the square is 1
```

From the signature of the accept function:

```
fn accept<V: ShapeVisitor>(&self, sv: &mut V);
```

We can see that static dispatch is used here. If we were to prefer dynamic dispatch, its signature would be:

```
fn accept(&self, sv: &mut ShapeVisitor)
```

As an exercise, rewrite the previous code in dynamic dispatch form (see `Chapter 6/exercises/visitor_pattern_dd.rs`).

Summary

In this chapter, we discovered the object-oriented nature of Rust, by defining methods on structs and implementing traits. Finally we came to see that traits are the structuring concept of Rust, and that using their full potential enables almost all object-oriented programming concepts.

In the following chapter, we will expose the crown jewels of the Rust language, which are the foundation of its memory safety behavior.

7
Ensuring Memory Safety and Pointers

This is probably the most important chapter in this book. Here we describe in detail the unique way in which the Rust borrow checker mechanism detects problems at compile time to prevent memory safety errors. It is fundamental to all else in Rust--the language is focused on these concepts of ownership and borrowing. Some of this material has already been discussed earlier, but here we deepen and strengthen that foundation.

We will cover the following topics:

- Pointers and references
- Ownership and borrowing
- Boxes
- Reference counting
- Overview of pointers

Trying out and experimenting with the examples is key here, as there are many concepts that you may not be familiar with yet.

Pointers and references

`Chapter 2`, *Using Variables and Types*, in section, *The stack and the heap* gave us the basic information we needed to understand memory layout in Rust. Let's recap here, and fill in some gaps.

Stack and heap

When a program starts up, by default a 2 Mb chunk of memory called the stack is granted to it. The program will use its stack to store all its local variables and function parameters, for example an `i32` variable takes 4 bytes on the stack. When our program calls a function, a new stack frame is allocated to it. Through this mechanism, the stack knows in which order functions are called, so that functions return correctly to the calling code, while possibly returning values.

Dynamically sized-types, like strings or vectors, can't be stored on the stack. For these values, a program can request memory space on its heap, which is a much bigger piece of memory than the stack.

When possible, stack allocation is preferred in Rust over heap allocation, because accessing the stack is a lot more efficient.

Lifetimes

All variables in Rust code have a lifetime, which is the area of code in which the variable is defined. Suppose we declare a variable `n` with the binding `let n = 42u32;` Such a value is valid from where it is declared to when it is no longer referenced; its lifetime ends there. This is illustrated in the following code snippet:

```
// see code in Chapter 7/code/lifetimes.rs
fn main() {
    let n = 42u32;
    let n2 = n; // a copy of the value from n to n2
    println!("The value of n2 is {}, the same as n", n2);
    let p = life(n);
    println!("p is: {}", p); // p is: 42
    println!("{}", m);  // error: unresolved name `m`.
    println!("{}", o);  // error: unresolved name `o`.
}

fn life(m: u32) -> u32 {
    let o = m;
    o
}
```

The lifetime of `n` ends when the `main()` function ends; in general the start and end of a lifetime happen in the same scope, here a function. The words lifetime and scope are synonymous, but we generally use the word lifetime for referring to the extent of a reference. As in other languages, local variables or parameters declared in a function do not exist anymore when the function has finished executing: in Rust we say their lifetime has ended. When the lifetime of a variable has ended, its memory is freed automatically, so the variable can no longer be used in the following code. This is the case for variables `m` and `o` in the code snippet mentioned earlier, which are only known in the function `life`.

Likewise the lifetime of a variable declared in a nested block is restricted to that block, like the `phi` variable in the following example:

```
{
   let phi = 1.618;
}
println!("The value of phi is {}", phi); // error
```

Trying to use the `phi` variable when its lifetime is over results in an error:

```
error: cannot find value `phi` in this scope`
```

The lifetime of a value can be indicated in code by an annotation and it is sometimes mandatory to do so. A lifetime is written as: `'a`, which reads as: lifetime a, where `a` is simply an indicator; it could also be written as `'b`, `'n` or `'life`. It's common to see single letters used to represent lifetimes. In the preceding example, an explicit lifetime indication was not necessary, since there were no references involved. All values tagged with the same lifetime have at its maximum that same lifetime.

We know this notation already from `'static`, which, as we saw in `Chapter 3`, *Using Functions And Control Structures* in the section *Functions*, is the lifetime of things that last for the entire length of the program.

In the following example we have a function `transform` which explicitly declares the lifetime of its parameter `s` of type `&str` to be `'a`:

```
fn transform<'a>(s: &'a str) { /* ... */ }
```



Note the `<'a>` indication right after the name of the function.

In nearly all cases this explicit indication is not needed because the compiler is smart enough to deduce the lifetimes, so we could simply write:

```
fn transform_without_lifetime(s: &str) { /* ... */ }
```

In fact, only things relating to references (such as a struct which contains a reference, or a function that has parameters that are a reference) need lifetimes.

Here is an example where, even when we indicate a lifetime specifier `'a`, the compiler does not allow our code. Suppose we have defined a struct `Magician` as:

```
struct Magician {
    name: &'static str,
    power: u32
}
```

Here `'static` is mandatory, if we leave it out, we get the error:

```
error[E0106]: missing lifetime specifier
```

If we define the following function:

```
fn return_magician<'a>() -> &'a Magician {
    let mag = Magician { name: "Gandalf", power: 4625};
  &mag
}
```

We get the following error:

```
error: `mag` does not live long enough
```

Why is this? The lifetime of the `mag` value ends when the function `return_magician` ends, but nevertheless this function tries to return a reference to that `Magician` value, which no longer exists. Such an invalid reference is known as a **dangling pointer**. This is a situation which clearly would lead to errors and cannot be allowed.

The lifetime of a pointer must always be shorter or equal than that of the value which it points to, thus avoiding dangling (or null) references.

In some situations the decision of whether the lifetime of an object has ended is complicated, but in almost all cases the borrow checker does this for us automatically by inserting lifetime annotations in the intermediate code, so we don't have to. This is known as **lifetime elision**.

For example, when working with structs, we can safely assume that the struct instance and its fields have the same lifetime. Only when the borrow checker is not completely sure it can happen do we need to indicate the lifetime explicitly, but this happens only on rare occasions, mostly when references are returned.

One example is when we have a struct with fields that are references. This code won't compile:

```
struct MagicNumbers {
    magn1: &u32,
    magn2: &u32
}
```

And gives us an error again:

```
error[E0106]: missing lifetime specifier
```

We have to change it to the following:

```
struct MagicNumbers<'a> {
    magn1: &'a u32,
    magn2: &'a u32
}
```

This is to specify that both the struct and the fields have the same lifetime `'a`.



Exercise:

Explain why the following code won't compile:

```
// see code in Chapter 7/exercises/dangling_pointer.rs:
fn main() {
let m: &u32 = { let n = &5u32; &*n };
let o = *m; }
```

Same question for this code snippet:

```
let mut x = &3; { let mut y = 4; x = &y; }
```



And this snippet:

```
struct IntNumber<'a> {
x: &'a i32,
}
let x = 1;
{ | let y = &5;
let f = IntNumber { x: y };
x = &f.x; } println!("{}", x);
```

Carefully read the error messages and the hints to what is wrong and how it could be solved.

Copying and moving values - The copy trait

In the code we looked at above (see `Chapter 7/code/lifetimes.rs`) the value of `n` is copied to a new location each time `n` is assigned via a new let binding or passed as a function argument:

```
let n = 42u32;
let n2 = n; // no move, only a copy of the value n to n2

life(n); // copy of the value n to m

fn life(m: u32) -> u32 {
    let o = m; // copy of the value m to o
    o
}
```

At a certain moment in the program's execution we would have four memory locations containing the copied value 42, which we could visualize as follows:

STACK

n	42
n2	42
m	42
o	42

Each value disappears (and its memory location is freed) when the lifetime of its corresponding variable ends, which is at the end of the function or code block in which it is defined. Nothing much can go wrong with this *Copy* behavior, in which the value (its bits) is simply copied to another location on the stack. Many built-in types, like `u32` and `i64` work like this, and this copy-value behavior is defined in Rust as the `Copy` trait, which `u32` and `i64` implement.

You can also implement the `Copy` trait for your own type, provided all of its fields or items implement `Copy`.

```
Let's now look at what happens when we do struct assignments:
struct MagicNumber {
    value: u64
}
```

Try the following code:

```
let mag = MagicNumber {value: 42};
let mag2 = mag;
let mag3 = mag;
```

The standard behavior is different, we now get a compiler error:

```
error[E0382]: use of moved value: `mag`

    let mag2 = mag;
        ---- value moved here
    let mag3 = mag;
        ^^^^ value used here after move
```



Note: Move occurs because `mag` has type `MagicNumber`, which does not implement the `Copy` trait

When `mag` was assigned to `mag2`, the ownership of that value was transferred to `mag2`--the value was moved. `mag` doesn't own it anymore, so using `mag`, for example in a new assignment cannot work, providing the following error:

```
value used after move
```

We'll go into more detail in the section *Ownership and Borrowing*.

Having only one owner of a value is a good thing: then the value can only be changed through one variable! This eliminates a lot of potential bugs.

One way to solve the compiler error is suggested in the note--`let MagicNumber` implement the `Copy` trait.

A type that does not implement the `Copy` trait, as is `MagicNumber` at this moment, is called **non-copyable**.

To make `MagicNumber` copyable, there is one more thing we need to know.

The `Clone` trait is a supertrait of `Copy` which states that every structure that implements `Clone` can duplicate itself. So if a struct implements `Copy`, it should also implement `Clone`. The `Clone` specifies that the following function must be implemented:

```
    fn clone(&self) -> Self;
```

It takes a reference to the current object (`&self`) and returns a duplicate of that, which is of course of the same type `Self`.

Let's now implement `Copy` for the struct `MagicNumber`, which contains a field of type `u64`. This can be done in two ways:

- One way is by explicitly naming the `Copy` and `Clone` implementation like this:

```
impl Copy for MagicNumber {}
      impl Clone for MagicNumber {
    fn clone(&self) -> MagicNumber {
        *self
    }
}
```

- The `*self` here is the dereferencing operator which says: return the value in the memory location `&self` points to.

- Or we can annotate the `struct` with an attribute and let the compiler do the work for us:

```
#[derive(Copy, Clone)]
struct MagicNumber {
    value: u64
}
```

Now our previous code is allowed:

```
let mag2 = mag;
let mag3 = mag;
```

The `mag`, `mag2`, and `mag3` variables are distinct copies because they have different memory addresses, which we can show as follows (the values shown will differ at each execution):

```
println!("address mag:  {:p}", &mag);  // address mag: 0x6ebbcff550
println!("address mag2: {:p}", &mag2); // address mag2: 0x6ebbcff558
println!("address mag3: {:p}", &mag3); // address mag3: 0x6ebbcff568
```

The `&mag` instance is the address of the `mag` value; we have to use `:p` in the format string, `p` stands for pointer.

Because `MagicNumber` now also implements the `Clone` trait, we can also use the `clone()` method explicitly to make a copy of `mag` to a different object `mag4`, like this:

```
let mag4 = mag.clone();
println!("address mag4: {:p}", &mag4); // address mag4: 0x7c0053f820
```



Remark:

If we want to display the contents of our struct like this:
`println!("mag is: {}", mag);`

Or:
`println!("mag is: {:?}", mag);`

We get an error:
`error[E0277]: the trait bound `MagicNumber: std::fmt::Display` is not satisfied;`

Rust tells us that it doesn't know how to display the struct's value. To let the `{:?}` version work, which is the debug format-string, `MagicNumber` has to implement the `Debug` trait, which we can do by adding a `#[derive(Debug)]` attribute.

(All three traits can also be combined in: `#[derive(Debug, Copy, Clone)]`).

Now the last print statement gives us the following output:

```
mag is: MagicNumber { value: 42 }
```

The following code snippet shows how to implement the `Clone` trait for a struct:

```
// see code in Chapter 7/code/clone.rs:
struct Block {
    number: Box<i32>
}

impl Clone for Block {
```

```
        fn clone(&self) -> Self {
            Block{ number: self.number.clone() }
        }
    }

    fn print_block(block: Block) {
        println!("{:p}: {:?}", block.number, block.number);
    }

    fn main() {
        let block = Block{ number: Box::new(1) };
        println!("{:p}: {:?}", block.number, block.number);

        print_block(block.clone());
    }
```

This prints out the following output:

```
0x20c5ca23b00: 1
0x20c5ca2cbe0: 1
```

Let's summarize

- In general the assignment `n = m` is a move, which means that `m` is not usable anymore-the content is moved from one owner to another and the old owner is invalidated.
- But when `m` is a primitive type or its type implements the `Copy` trait, it is still usable afterwards. This does not change what the assignment does, only whether you are allowed to use the old object.
- You can force copy semantics when you need it by using `n = m.clone()`.

Pointers

The variable `n` in the binding `let n = 42i32;` is stored on the stack. Values on the stack or on the heap can be accessed by pointers. A pointer is a variable that contains the memory address of some value. To access the value it points to, dereference the pointer with `*`. This happens automatically in simple cases like in `println!` macro or when a pointer is given as a parameter to a method. For example in the following code `m` is a pointer containing the address of `n`:

```
// see code in Chapter 7/code/references.rs:
let m = &n;
```

```
println!("The address of n is {:p}", m);
println!("The value of n is {}", *m);
println!("The value of n is {}", m);
```

The preceding code prints out the following output. The address of the variable `n` differs with each program run:

```
The address of n is 0x7078eff8bc
The value of n is 42
The value of n is 42
```

Why do we need pointers? When we work with dynamically allocated values that can change in size like a string, the memory address of that value is not known at compile time. Because of this, the memory address needs to be calculated at runtime. So to be able to keep track of it, we need a pointer to it, whose value changes when the string's location in memory changes.

The compiler automatically takes care of the memory allocation of pointers, and the freeing up of memory when their lifetime ends. You don't have to do this yourself like in C/C++, where you could screw up by freeing at the wrong moment, or freeing memory multiple times.

This incorrect use of pointers in languages like C++ leads to all kinds of problems.

However, Rust enforces a strong set of rules at compile time called the *borrow checker*, so we are protected against them. We have already seen them in action, but from here onwards we'll explain the logic behind its rules.

Pointers can also be passed as arguments to functions, and can be returned from functions, but the compiler severely restricts their usage.

When passing a pointer value to a function, it is always better to use the reference-dereference `&*` mechanism, like in this example:

```
let q = &42;
   println!("{}", square(q)); // 1764

fn square(k: &i32) -> i32 {
    *k * *k
}
```

This prints out the following output:

```
The square is: 1764
```

Rust has many kinds of pointers, which we will explore in this chapter. All pointers (except raw pointers, discussed in `Chapter 10`, *Programming at the Boundaries*) are guaranteed to be non-null (that is, they point to a valid location in memory) and are automatically cleaned up.

References

In our previous example, `m` which had the value `&n` is the simplest form of pointer, called a **reference** (or **borrowed pointer**)--the variable `m` is a reference to the stack-allocated variable `n` and has type `&i32` because it points to a value of type `i32`.



In general, when `n` is a value of type `T`, then the reference `&n` is of type `&T`.

Here `n` is immutable, so `m` is also immutable; trying to change the value of `n` through `m` with `*m = 7;` for example, results in an error:

```
cannot assign to immutable borrowed content `*m`
```

Rust does not let you change an immutable variable via its pointer, contrary to C.

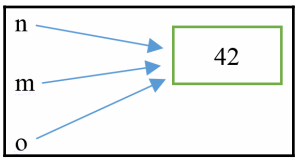
Because there is no danger of changing the value of `n` through a reference, multiple references to an immutable value are allowed, they can only be used to read the value, for example:

```
let o = &n;
println!("The address of n is {:p}", o);
println!("The value of n is {}", *o);
```

Printing out as before:

```
The address of n is 0x7078eff8bc
The value of n is 42
```

We could represent this situation in memory like this:



It will be clear that working with pointers like this or in much more complex situations necessitates much stricter rules than the `Copy` behavior. For example, the memory can only be freed when there are no variables or pointers associated with it anymore. And when the value is mutable, can it be changed through any of its pointers? These stricter rules, described by the ownership and borrowing system from the next section, are enforced by the compiler.

Mutable references do exist, and they are declared as:

```
let m = &mut n
```

However the variable `n` also has to be a mutable value. When `n` is immutable, the compiler rejects the `m` mutable reference binding with the error:

```
error: cannot borrow immutable local variable `n` as mutable
```

This makes sense, as immutable variables cannot be changed, even when you know their memory location.

To reiterate, in order to change a value through a reference, both the variable and its reference have to be mutable, like in the following code snippet:

```
    let mut u = 3.14f64;
    let v = &mut u;
    *v = 3.15;
println!("The value of u is now {}", *v);
```

This will print the following output:

```
The value of u is now 3.15
Now the value at the memory location of u is changed to 3.15
```

But note that we now cannot change (or even print) that value anymore by using the variable `u`:

A statement like `u = u * 2.0;` or even a simple `println!("The value of u is {}", u);` give us respectively the compiler error:

```
errors: cannot assign to `u` because it is borrowed
```

(We explain why this is so in the following section on *Ownership and Borrowing*)

```
cannot borrow `u` as immutable because it is also borrowed as mutable
```

We say that borrowing a variable (by taking a reference to it, here `v`) freezes that variable, the original variable `u` is frozen (no longer usable), until the `v` reference goes out of scope.

Also we can only have one mutable reference: `let w = &mut u;` which results in the following error:

```
error: cannot borrow `u` as mutable more than once at a time
```

The compiler even adds the following note to the previous code line with: `let v = &mut u;` which results in the following error:

```
note: previous borrow of `u` occurs here; the mutable borrow prevents
subsequent moves, borrows, or modification of `u` until the borrow ends.
```

This is logical--the compiler is (rightfully) concerned that a change to the value of `u` through one reference might change its memory location, because the variable `u` might change in size (in case of a more complex variable than an integer) and so not fit anymore in its previous location, and would have to be relocated to another address. This would render all other references to `u` invalid, and even dangerous, because through them we might inadvertently change another variable that has taken up the `u` variable's previous location!

A mutable value can also be changed by passing its address as a mutable reference to a function, like in this example:

```
let mut m = 7;
add_three_to_magic(&mut m);
println!("m is now {}", m);  // prints out m is now 10
```

With the function `add_three_to_magic` declared as:

```
fn add_three_to_magic(num: &mut i32) {
    *num += 3;  // value is changed in place through +=
}
```



To summarize, when `n` is a mutable value of type `T`, then only one mutable reference to it (of type `&mut T)` can exist at any time. Through this reference, the value can be changed.

References are frequently used as function parameters: they avoid moving (which means in most cases copying) the value to the function, which could decrease performance when the value is large. Instead passing a reference only copies the address of the value to the function, not the value, thereby saving unnecessary memory usage. Consider a string buffer that contains a large amount of data. Copying that around will cause the program to be much slower.

Match, struct, and ref

If you want to get a reference to a matched variable inside a `match`, use the `ref` keyword, as in the following example:

```
// see code in Chapter 7/code/ref.rs
fn main() {
   let n = 42;
   match n {
      ref r => println!("Got a reference to {}", r),
   }

   let mut m = 42;
   match m {
      ref mut mr => {
         println!("Got a mutable reference to {}", mr);
         *mr = 43;
      },
   }
   println!("m has changed to {}!", m);
}
```

This prints out the following output:

```
Got a reference to 42
Got a mutable reference to 42
m has changed to 43!
```

The `r` variable inside the `match` has the type `&i32`. In other words, the `ref` keyword creates a reference for use in the pattern. If you need a mutable reference, use `ref mut`.

We can also use `ref` to get a reference to a field of a struct or tuple in a destructuring via a `let` binding. For example, while reusing the struct `Magician` we can extract the name of `mag` by using `ref` and then return it from the `match`:

```
let mag = Magician { name: "Gandalf", power: 4625};
let name = {
    let Magician { name: ref ref_to_name, power: _ } = mag;
    *ref_to_name
};
println!("The magician's name is {}", name);
```

This prints out the following output:

```
The magician's name is Gandalf.
```

References are the most common pointer type and have the most possibilities; other pointer types should only be applied in very specific use-cases.

Ownership and borrowing

In the previous section the word `borrowed` was mentioned in most error messages. What's this all about? What is the logic behind this borrow checker mechanism?

Every program, whatever it does, like reading data from a database or making a computation, is about handling resources. The most common resource in a program is the *memory space* allocated to its variables. Other resources could be files, network connections, database connections, and so on.

Ownership

Every resource is given a name when we make a *binding* to it with `let`; in Rust speak we say that the resource gets an *owner*. For example, in the following code snippet `klaatu` owns the piece of memory taken up by the `Alien` struct instance:

```
// see code in Chapter 7/code/ownership1.rs
struct Alien {
    planet: String,
    n_tentacles: u32
}

fn main() {
    let mut klaatu = Alien{ planet: "Venus".to_string(),
    n_tentacles: 15 };
}
```

Only the owner can change the object it points to, and there can only be one owner at a time, because the owner is also responsible for freeing the object's resources. This makes sense; if an object could have many owners, its resources could be freed more than once, which would lead to problems. When the owner's lifetime has passed, the compiler frees the memory automatically.

Moving a value

The owner can move the ownership of the object to another variable like this:

```
let kl2 = klaatu;
```

Here the *ownership has moved* from `klaatu` to `kl2`, but no data is actually copied. The original owner `klaatu` cannot be used anymore:

```
println!("{}", klaatu.planet);
```

The compiler gives the following error:

```
error: use of moved value 'klaatu.planet'.
```

The same thing happens if, after the assignment to `kl2`, we pass `klaatu` as a parameter to a function `transform`:

```
let kl2 = transform(klaatu);
fn transform(a: Alien) -> Alien {
   Alien { planet:"Jupiter".to_string(), n_tentacles:0 }
}
```

This gives the following error:

```
error: use of moved value: `klaatu`
```

The `use of moved value` error goes away if we assign the return value of the function to `klaatu` itself:

```
let klaatu = transform(klaatu);
println!("{}", klaatu.planet); // Jupiter
```

But this is not a very elegant solution.

Borrowing a value

On the other hand we can borrow the resource by making a (mutable) reference `kl2` to `klaatu` with:

```
let kl2 = &klaatu;         // a borrow or reference
```

Or:

```
let kl2 = &mut klaatu; // a mutable borrow or reference
```

A borrow is a temporary reference, passing the address of the data structure through `&`. The borrowed reference can access the contents of the memory location, but does not own the value. The owner gives up full control over a borrowed value while it is being borrowed.

Now, in the case of the mutable borrow, `kl2` can change the object, for instance when our alien loses a tentacle in a battle:

```
kl2.n_tentacles = 14;
println!("{} - {}", kl2.planet, kl2.n_tentacles);
```

This prints out the following output:

```
Venus - 14
```

But if we try to change the alien's planet through `klaatu`, we can do it as follows:

```
klaatu.planet = "Pluto".to_string();
```

We get the following error:

```
error: cannot assign to `klaatu.planet` because it is borrowed
```

It was indeed borrowed by the `kl2` reference.

Like in everyday life, while an object is borrowed, the owner does not have access to it as it is no longer in its possession. In order to change the resource, `klaatu` needs to own it, without the resource being borrowed at the same time.

Rust even explains this to us with the note it adds:

```
    borrow of `klaatu.planet` occurs here ownership.rs:18    let kl2 = &mut
klaatu;
```

Because `kl2` borrows the resource, Rust also even forbids us to access the instance with its former name `klaatu`:

```
println!("{} - {}", klaatu.planet, klaatu.n_tentacles);
```

The compiler then throws the following error:

```
    error: cannot borrow `klaatu.planet` as immutable because `klaatu` is
also borrowed as mutable.
```

When a resource is moved or borrowed, the (original) owner can no longer use it. This prevents the memory problem that is known as a **dangling pointer**, which is the use of a pointer that points to an invalid memory location.

But here is a revelation: if we isolate the borrowing by `kl2` in its own block, like in:

```
// see code in Chapter 7/code/ownership2.rs
fn main() {
    let mut klaatu = Alien{ planet: "Venus".to_string(),    n_tentacles: 15
};
    {
      let kl2 = &mut klaatu;
      kl2.n_tentacles = 14;
      println!("{} - {}", kl2.planet, kl2.n_tentacles);
// prints: Venus - 14
    }
}
```

The former problems have disappeared! After the block `kl2` can no longer be used, but we can now do for example:

```
println!("{} - {}", klaatu.planet, klaatu.n_tentacles);
klaatu.planet = "Pluto".to_string();
println!("{} - {}", klaatu.planet, klaatu.n_tentacles);
```

This prints the following output:

```
Venus - 10
Pluto - 10
```

Why is this? Because after the closing `}` of the code block in which the `kl2` reference was bound, its lifetime ended. The borrowing was over (a borrow has to end sometime) and `klaatu` reclaimed full ownership, and thus the right to change. When the compiler detects that the lifetime of the original owner `klaatu` eventually ends, the memory occupied by the struct instance is automatically freed.

In fact this is a general rule in Rust: whenever an object goes out of scope and it doesn't have an owner anymore, its destructor is automatically called and the resources owned by it are freed, so that there never can be any memory (or other resource) leaks. This is described in technical terms as Rust obeys the **Resource Acquisition Is Initialization** (**RAII**) rule; for more info see
`http://en.wikipedia.org/wiki/Resource_Acquisition_Is_Initialization`.

As we experimented in the previous section on *References*, a resource can be immutably borrowed many times, but while immutably borrowed, the original data can't be mutably borrowed.

Implementing the Drop trait

The following code snippet in which the `Drop` trait is implemented for a struct clearly shows the moment when the value `block` is freed:

```
// see code in Chapter 7/code/drop.rs
struct Block {
    number: i32
}

impl Drop for Block {
    fn drop(&mut self) {
        println!("Dropping!");
    }
}

fn print_block(block: Block) {
    println!("In function print_block");
}

fn main() {
    let block = Block{ number: 1 };
    // move of value block:
    print_block(block);
    println!("Back in main!");
}
```

This prints out the following output:

```
In function print_blockDropping!Back in main!
```

So the `block` reference is freed at the end of the `print_block` function.

Moving closure

In the previous chapter we introduced the concept of a moving closure, which takes ownership of its variables. The following code illustrates clearly the difference between a normal closure and a moving closure:

```
// see code in Chapter 7/code/moving_closure.rs
struct Block {
    number: i32
}

fn main() {
    let block = Block{ number:1 };
    // ordinary closure:
    let closure = || { println!("n: {:?}", block.number); };
    closure();
    println!("n: {:?}", block.number);

    let block = Block{ number:1 };
    // moving closure:
    // closure takes ownership of the block value
    let closure = move || {println!("n: {:?}", block.number); };
    closure();
    // error: use of moved value: `block.number`
    // println!("n: {:?}", block.number);
}
```

This prints out the following on three consecutive lines:

```
n: 1
```

After the moving closure is defined, the `block` reference can no longer be accessed.

Another way to move a resource (thus transferring the ownership) is to pass it as argument to a function; try this out in the following exercise:

Exercises:

Examine the situation when `kl2` is not a mutable reference `let kl2 = &klaatu;`.
Can you change the instance through `kl2`?
Can you change the instance through `klaatu`?
Explain the error with what you know about ownership and borrowing (see `Chapter 7/exercises/ownership3.rs`).
What happens if in the previous program we do `let klaatuc = klaatu;` before we define the binding `let kl2 = &klaatu;`?



Examine if you can change the mutability of a resource by moving from an immutable owner to a mutable owner.
For our struct Alien, write a method `grow_a_tentacle` that increases the number of tentacles by one (see `Chapter 7/exercises/grow_a_tentacle.rs`).
Construct a vector `p` containing `[1, 2, 3]`.
Then make a function `increment` that returns a copy of p with all values incremented.
Now make a function `increment_mut` that increments all values of p in place. (see `Chapter 7/exercises/increment_vector.rs`)
Some exercises with mutable pointers (see `Chapter 7/exercises/pointer_mutability.rs`).

Boxes

Another pointer type in Rust is called the boxed pointer `Box<T>`, which can be defined for a value of a generic type `T`. A box is a non-copyable value. This pointer type is used to allocate objects on the heap.

```
For example, here we allocate an Alien value on the heap with:
// see code in Chapter 7/code/boxes1.rs
let mut a1 = Box::new(Alien{ planet: "Mars".to_string(), n_tentacles: 4 });
println!("{}", a1.n_tentacles); // 4
```

The mutable variable `a1` is the only owner of this memory resource that may read from and write to it.

We can make a reference to the value pointed to by the box pointer, and if both the original box and this new reference are mutable, we can change the object through this reference:

```
let a2 = &mut a1;
println!("{}", a2.planet ); // Mars
a2.n_tentacles = 5;
```

After such a borrow the usual ownership rules as above hold: `a1` no longer has access, not even for reading:

```
// error: cannot borrow `a1.n_tentacles` as immutable because `a1` is also
borrowed as mutable
println!("{}", a1.n_tentacles); // is error!
// error: cannot assign to `a1.planet` because it is borrowed
a1.planet = "Pluto".to_string();  // is error!
```

We can also use this mechanism to put simple values on the heap, like this:

```
let n = Box::new(42);
```

As always `n` points by default to an immutable value, and any attempt to change this with:

```
*n = 67;
```

Provokes the following error:

```
error: cannot assign to immutable `Box` content `*n`.
```

As usual we dereference a `Box` with `*`:

```
let p = *n;
println!("{}", p); // 42
```

Another reference can also point to the dereferenced `Box` value:

```
let q = &*n;
println!("{}", q); // 42
```

In the following example we see again a boxed value pointed to by n, but now the ownership of the value is given to a mutable pointer m:

```
// see code in Chapter 7/code/boxes2.rs
let n = Box::new(42);
let mut m = n;
*m = 67;
// println!("{}", n); // error: use of moved value: `n`
println!("{}", m); // 67
```

By dereferencing the `m` variable and assigning a new value, this value is entered into the memory location originally pointed to by the variable `n`. Of course the `n` variable cannot be used anymore, we get the following error:

```
error: use of moved value: `n`
```

Because the variable `n` is no longer the owner of the value.

```
Here is another example where the ownership clearly has moved from a1 to
a2:
let mut a1 = Box::new(Alien{ planet: "Mars".to_string(), n_tentacles: 4 });
   let a2 = a1;
   println!("{}", a2.n_tentacles); // 4
```

There is no data being copied here, only the address of the struct value. After the move `a1` can no longer be used to access the data, the variable `a2` is responsible for freeing the memory.

If `a2` is given as argument to a function like `use_alien` below, `a2` in its turn gives up the ownership, which is transferred to the function:

```
   use_alien(a2);
   // Following line gives the error: use of moved value: `a2.n_tentacles`
   // println!("{}", a2.n_tentacles);
} // end of main() function

fn use_alien(a: Box<Alien>) {
    println!("An alien from planet {} is freed after the closing brace.",
a.planet);
}
```

This prints out the following output:

```
An alien from planet Mars is freed after the closing brace.
```

Indeed when the `use_alien()` function is finished executing, the memory allocation for that value is freed.

In general let your function always take a simple reference as parameter (as in the `square` function mentioned earlier in section *Pointers*), rather than a parameter of the `Box` type. We could improve our example by calling a function `use_alien2` as follows:

```
fn use_alien2(a: &Alien) {
   println!("An alien from planet {} is freed", a.planet);
}
```

And calling it like this:

```
use_alien2(&*a2);
```

Boxes also allow for automatic dereferencing, as in this following code snippet:

```
let ua = Box::new([1, 2, 3]);
println!("{}", ua[0]);
```

Which prints the following output:

```
1
```

Sometimes your program needs to manipulate a recursive data structure that refers to itself, like the following struct

```
struct Recurs {
    list: Vec<u8>,
    rec_list: Option<Box<Recurs>>
}
```

In the codefile `linked_list.rs`, you can find a working example of a recursive linked list datastructure:

```
enum List {
    Cons(u32, Box<List>),
    Nil,
}
```

A linked list node `List` can take on any of these two variants:

- `Cons`: Which is a tuple struct that wraps an element of type `u32` and a `Box` pointer to the next node
- `Nil`: Which is a node that signifies the end of the linked list

```
Especially note the 2 recursive methods like len and stringify.
```

This represents a list of lists of bytes. The `rec_list` variable is either a `Some<Box<Recurs>>` containing a `Box` pointer to another list, or a `None` value which means the list of lists ends there. Because the number of items in this list (and thus its size) is only known at runtime, such structures always must be constructed as a `Box` type.

For other use-cases, prefer references over Boxes.

Reference counting

Sometimes you need several references to an immutable value at the same time, this is also called **shared ownership**. The pointer type `Box<T>` can't help us out here, because this type has a single owner by definition. For this, Rust provides the generic reference counted box `Rc<T>`, where multiple references can share the same resource. The `std::rc` module provides a way to share ownership of the same value between different `Rc` pointers; the value remains alive as long as there is least one pointer referencing it.

In the following example, we have Aliens that have a number of tentacles. Each `Tentacle` is also a struct instance and has to indicate to which `Alien` it belongs; besides that it has other properties like a degree of poison. A first attempt at this could be the following code which however does not compile:

```
// see Chapter 7/code/refcount_not_good.rs):
struct Alien {
    name: String,
    n_tentacles: u8
}

struct Tentacle {
    poison: u8,
    owner: Alien
}

fn main() {
let dhark = Alien { name: "Dharkalen".to_string(), n_tentacles: 7 };

    // defining dhark's tentacles:
    for i in 0u8..dhark.n_tentacles {
        Tentacle { poison: i * 3, owner: dhark }; // <- error!
    }
}
```

The compiler gives a following error for the line in the `for` loop:

```
    error: use of moved value 'dhark' – move occurs because `dhark` has
type `Alien`, which does not implement the `Copy` trait
```

Each `Alien Tentacle` when it is defined seemingly tries to make a copy of the `Alien` instance as its owner, which would make no sense and is not allowed.

The correct version defines the owner in the `Tentacle` struct to have the type `Rc<Alien>`:

```
// see code in Chapter 7/code/refcount.rs
use std::rc::Rc;

#[derive(Debug)]
struct Alien {
    name: String,
    n_tentacles: u8
}

#[derive(Debug)]
struct Tentacle {
    poison: u8,
    owner: Rc<Alien>
}

fn main() {
    let dhark = Alien { name: "Dharkalen".to_string(), n_tentacles: 7 };

    let dhark_master = Rc::new(dhark);

    for i in 0u8..dhark_master.n_tentacles {
        let t = Tentacle { poison: i * 3, owner: dhark_master.clone() };
       println!("{:?}", t);
    }
}
```

This prints the following output:

```
Tentacle {poison: 0, owner: Alien{name: "Dharkalen", n_tentacles: 7}}
Tentacle {poison: 3, owner: Alien{name: "Dharkalen", n_tentacles: 7}}
Tentacle {poison: 6, owner: Alien{name: "Dharkalen", n_tentacles: 7 }}
Tentacle {poison: 18, owner: Alien{name: "Dharkalen", n_tentacles: 7}}
```

We envelop our Alien instance in an `Rc<T>` type with `Rc::new(dhark)`. Applying the `clone()` method on this `Rc` object provides each `Tentacle` with its own reference to the `Alien` object. Note that `clone()` here copies the `Rc` pointer, not the `Alien` struct. We also annotate the structs with `#[derive(Debug)]`, so that we can print out their instances through a `println!("{:?}", t);`.

If we want mutability inside our `Rc` type, we have to use either a `Cell` pointer if the value implements the `Copy` trait, or a `RefCell` pointer otherwise. Both these smart pointers are found in the `std:cell` module.

The `Rc` pointer type can however only be used inside one thread of execution only. If you need shared ownership across multiple threads, you need to use the `Arc<T>` pointer (atomic reference counted box), which is the thread-safe counterpart of `Rc` (see `Chapter 9</span>`, *Concurrency - Coding for Multicore Execution* in the section *Atomic reference counting*).

Overview of pointers

In the following table we summarize the different pointers used in Rust. `T` represents a generic type. We haven't yet encountered the `Arc`, `*const` and `*mut` pointers, but they are included here for completeness.

Notation	**Pointer type**	**What can this pointer do?**
`&T`	Reference	Allows one or more references to read `T`
`&mut T`	Mutable Reference	Allows a single reference to read and write `T`
`Box<T>`	Box	Heap allocated `T` with a single owner that may read and write `T`.
`Rc<T>`	`Rc` pointer	Heap allocated `T` with many readers
`Cell<T>`		Shared mutable memory location with `Copy` implemented
`RefCell<T>`		Mutable memory location
`Arc<T>`	`Arc` pointer	Same as above, but safe mutable sharing across threads (see `Chapter 8`, *Concurrency - Coding for Multicore Execution*)
`*const T`	Raw pointer	Unsafe read access to `T` (see `Chapter 10`, *Programming at the Boundaries*)
`*mut T`	Mutable raw pointer	Unsafe read and write access to `T` (see also `Chapter 10`, *Programming at the Boundaries*)

Summary

In this chapter we learned the intelligence behind the Rust compiler, embodied in the principles of ownership, moving values and borrowing--it is called the **borrow-checker** for a reason. We saw the different pointers Rust advocates--references, boxes, and reference counters. Now that we have a grasp on how this all works together, we will understand much better the errors, warnings, and messages the compiler may throw at us.

In the following chapter we will expose the bigger units of code organization in code like modules and crates, and how we can write macros to make coding less repetitive.

8

Organizing Code and Macros

This chapter starts with discussing the large-scale code-organizing structures in Rust, namely modules and crates. ;After working through the material, you'll understand the structure of Rust projects, and be able to build a structured app as well. More specifically, we look at the following topics:

- Building crates
- Defining a module
- Visibility of items
- Importing modules and file hierarchy
- Importing external crates
- Exporting a public interface
- Adding external crates to a project
- Working with random numbers

We will touch upon how to build macros in order to generate code and save time and effort, particularly in these topics:

- Why macros?
- Developing macros
- Using macros from crates
- Some other built-in macros

Modules and crates

Until now, we only looked at a situation where our code fitted in one file. But when a project evolves, we will want to split the code into several files, for example, by putting all data structures and methods that describe certain functionality in the same file. How will the main code file be able to call these functions in other files?

Also, when we start getting multiple functions in various files, it sometimes happens that we want to use the same name for two different functions. How can we properly differentiate between such functions, and how can we make it so that some functions are callable everywhere, and others are not? For this, we need what other languages call namespaces and access modifiers; in Rust this is done through the *module system*.

Building crates

At the highest level, there is the crate. The Rust distribution contains a number of crates, such as the `std` crate of the standard library, which we have already used often. Other built-in crates include the `collections` crate, with functionality to work with strings, vectors, linked lists, and key-value maps, and the `test` or `rustc-test` crates, with unit-testing and micro-benchmarking functionality.

A crate is the equivalent of a package or library in other languages. It is also the unit of compilation: `rustc` only compiles one crate at a time. What does this mean? When our project has a code file containing a `main()` function, then it is clear that our project is an executable program (also called a binary), starting execution in `main()`. For example, if we compile `structs.rs` as follows:

```
rustc structs.rs
```

A `.exe` file, `structs.exe`, is produced in Windows, which can be executed on its own (and equivalent `structs` executables on other operating systems). This is the standard behavior when invoking rustc. When working with Cargo (see `Chapter 1`, *Starting with Rust*) we have to indicate that we want a binary project at its creation with the `--bin` flag:

```
cargo new projname --bin
```

However, often you want to write a project whose code will be called from other projects, a so-called **shared library** (in Windows this is a `.dll` file, in Linux an `.so` file, and on OSX a `.dylib` file). In that case, your code will only contain data structures and functions to work on them, so you must explicitly indicate this to the compiler using the `--crate-type` flag with the `lib` option:

```
rustc --crate-type=lib  structs.rs
```

The resulting file is far smaller in size and is called `libstructs.rlib`; the suffix is now `.rlib` (for Rust library) and `lib` is prepended before the filename. If you want the crate to have another name, like `mycrate`, then use the `--crate-name` flag as follows:

```
rustc --crate-type=lib  --crate-name=mycrate  structs.rs
```

This creates a `libmycrate.rlib` as the output file.

An alternative to using the `rustc` flags is to put this info as the attributes at the top of the code file, like this:

```
// from Chapter 8/code/structs.rs
#![crate_type = "lib"]
#![crate_name = "mycrate"]
```

The `crate_type` attribute can take the following values: `bin`, `lib`, `rlib`, `dylib`, and `staticlib`, according to whether you want a binary executable or a library of a certain type, dynamically or statically linked (in general when an `attr` attribute applies to a whole crate, the syntax to use in code is `#![crate_attr]`).

Each library used in an application is a separate crate. In any case, you need an executable (binary) crate that uses the library crates.

Rust has a built-in library manager called Cargo (for more info on Cargo, see `Chapter 1`, *Starting with Rust*, and the *Working with Cargo* section); it creates a library project by default. Typically, when starting your own project, you would create it with Cargo, use the code organization techniques you will learn in this chapter, and then make this code into an executable with:

```
cargo build
```

You can install other crates into your project from the crates repository `https://crates.io`; in the *Adding external crates to a project* section, we see how this is done.

Defining a module

Crates are the compiled entities that get distributed on machines to execute. All the code of a crate is contained in an implicit root module. This code can then be split up by the developer into code units called **modules**, which in fact form a hierarchy of sub-modules under the root module. That way the organization of our code can be greatly improved. An evident candidate for a module is the tests code, which we discussed in `Chapter 3`, *Using Functions and Control Structures* in the *The tests module* section.

Modules can also be defined inside other modules, as so-called **nested modules**. Modules do not get compiled individually; only crates get compiled. All module code is inserted into the crate source file before compilation starts.

In previous chapters, we used built-in modules from the `std` crate, such as `io`, `str`, and `vec`. The `std` crate contains many modules and functions that are used in real projects; the most common types, traits, functions, and macros (such as `println!`) are declared in the `prelude` module.

A module typically contains a collection of code items such as traits, structs, methods, other functions, and even nested modules. The module's name defines a *namespace* for all objects it contains. We define a module with the `mod` keyword and a lowercase name (like `game1`) as follows:

```
mod game1 {
   // all of the module's code items go in here
}
```

Similar to Java, each file is a module, and for every code file the compiler defines an implicit module, even when it does not contain the `mod` keyword. As we will see in the *Importing modules and file hierarchy* section, such a code file can be imported in the current code file with `mod filename;`.

Suppose `game1` is the name of a module that contains a function, `func2`. If you want to use this function in code external to this module, you would address it as `game1::func2`. But whether this is possible depends on the visibility of `func2`.

Visibility of items

Items in a module are by default only visible in the module itself; they are private to the module they are defined in. If you want to make an item callable from code external to the module, you must explicitly indicate this by prefixing the item with `pub` (which stands for public). In the following code, trying to call `func1()` is not allowed by the compiler--`error: function `func1` is private`:

```
// from Chapter 8/code/modules.rs
mod game1 {
    // all of the module's code items go in here
    fn func1() {
        println!("Am I visible?");
    }

    pub fn func2() {
        println!("You called func2 in game1!");
    }
}

fn main() {
    // game1::func1(); // <- error!
    game1::func2();
}
```

Calling `func2()` works without any problem because it is public, and this prints out the following output:

```
You called func2 in game1!
```

A function in a nested module can only be called if it is public, and if the nested module itself is declared public, like in this code snippet:

```
mod game1 {
    // other code
    pub mod subgame1 {
      pub fn subfunc1() {
         println!("You called subfunc1 in subgame1!");
      }
    }
}

fn main() {
    // other code
    game1::subgame1::subfunc1();
}
```

This prints out the following output:

```
You called subfunc1 in subgame1!
```

A function in a module must be prefixed with its module name when called. This distinguishes it from another function with the same name so that no name conflict can occur.

When a struct is accessed from outside the module in which it is defined, it is only visible when it is declared with `pub`. Moreover, its fields are private by default, so you have to explicitly declare as public those fields that you want to be visible outside. This is the **encapsulation property** (also called information hiding) from traditional object-oriented languages. In the following example, the fields `name` and `age` of struct `Magician` belong to the public interface, but `power` does not:

```
pub struct Magician {
    pub name: String,
    pub age: i32,
    power: i32
}
```

So, this statement:

```
let mag1 = game1::Magician { name: "Gandalf".to_string(), age: 725, power:
98};
```

Leads to the compiler error:

```
error: field `power` of struct `game1::Magician` is private.
```

To summarize, Rust has only two visibility modifiers:

- The default, which is only visible within the current module
- `pub`, which is exported to the parent module



Exercise:

Does this mean we cannot make instances from a struct with private fields?
Try to think of a way around this. (Hint: think about a constructor-like `new` function, see `Chapter 8/exercises/priv_struct.rs`).

Generalizing this way of reasoning, here you can see a general way to call a private method `f` on a private field `m_impl`, the so-called **Private Implementation Pattern** (**PIMPL**):

```
// see Chapter 8/code/mod_private.rs
mod library {
  pub struct Interface {
    m_impl: Impl    // private field
  }
  impl Interface {
    pub fn new() -> Interface {
      Interface{ m_impl: Impl }
    }
    pub fn f(&self){
      self.m_impl.f();
    }
  }
  struct Impl;
  impl Impl {
    fn f(&self){   // private method
      println!("f");
    }
  }
}

fn main() {
  let o = library::Interface::new();
  o.f();
}
```

This prints out `f`.

Importing modules and file hierarchy

The `use` keyword like in `use game1::func2;` imports a `func2` function from the `game1` module, so that it can simply be called with its name, `func2()`. You can even give it a shorter name with `use game1::func2 as gf2;` so that it can be called as `gf2()`.

When the module `game1` contains two (or more) functions, `func2` and `func3`, that we want to import, this can be done with `use game1::{func2, func3};`.

If you want to import all (`public`) items of the module `game1` you can do it with `*`:

```
use game1::*;
```

However, using such a global import is not best practice, except in modules for testing. The main reason is that a global import makes it harder to see where names are bound. Furthermore, they are forward-incompatible, since new upstream exports can clash with existing names.

Inside a module, `self::` and `super::` can be prepended to a path, like `game1::func2`, to distinguish, respectively, between a function in the current module itself and a function in the parent scope outside of the module:

```
self::game1::func2();
::game1::func2(); // same thing as self::
super::game1::func2();
```

The `use` statements are preferably written at the top of the code file, so that they work for the whole of the code. If you prepend the `use` statement with a `pub`, then you export these definitions (in other words, you make them visible) to the `super` level, like this:

```
pub use game1::{func2, func3};
```

If the code is inside the current module, write this as:

```
use self::game1::{func2, func3};
```

In the previous example, the module was defined in the main source file itself; in most cases a module will be defined in another source file. How do we import such modules? In Rust, we can insert the entire contents of a module source file into the current file by declaring the module at the top of the code (but after any `use` statements) like this--`mod modul1;` optionally preceded by `pub`.

This statement will look for a file, `modul1.rs`, in the same folder as the current source file, and import its code in the current code inside a module, `modul1`. If a `modul1.rs` file is not found, it will look for a `mod.rs` file in a subfolder `modul1`, and insert its code.

If we write it as `pub mod modul1`, then everything from `modul1` is accessible.

This is Rust's mechanism for constructing a main source file that imports the code of many other source files and can refer to them. The mechanism works in a hierarchical fashion, allowing the creation of complete trees containing source code.

Here is a simple example; `import_modules.rs` contains the following code:

```
// from Chapter 8/code/import_modules.rs
mod modul1;
mod modul2;
```

```
fn main() {
    modul1::func1();
    modul2::func1();
}
```

In a subfolder, `modul1`, we have the `mod.rs` file, containing this:

```
pub fn func1() {
    println!("called func1 from modul1");
}
```

The `modul2.rs` file in the same folder as `import_modules.rs` contains this:

```
pub fn func1() {
    println!("called func1 from modul2");
}
```



Note that these module source files don't contain the `mod` declaration anymore, because they were already declared in `import_modules.rs`.

Executing `import_modules` prints out the following:

```
called func1 from modul1
called func1 from modul2
```



Remark: ;

Compile `import_modules.rs` on the command-line with rustc, using the Sublime Text plugin, for example, it is not able to find the module code.

What happens if you simply call `func1()` in `main()`? Now the compiler doesn't know which `func1` to call, either from `modul1` or from `modul2`, resulting in `error: unresolved name `func1`.` But if we add `usemodul1::func1`, then calling `func1()` works and the ambiguity is resolved.

Importing external crates

In `Chapter 6`, *Using Traits and OOP in Rust*, in the *Traits* section, we developed in `traits.rs` structs for `Alien`, `Zombie`, and `Predator` characters that implemented a trait, `Monster`. The code file contained a `main()` function to make it executable. We will now incorporate this code (without the `main()` part) in a library project called `monsters`, and see how we can call this code.

Create the library project with `cargo new monsters`, creating a folder structure and a `monsters/src/lib.rs` file with template code:

```
#[test]
fn it_works() {
}
```

Remove this code and replace it with the code from `Chapter 6/code/traits.rs`, but omit the `main()` function. We also add a simple `print_from_monsters()` function to test calling it from the library:

```
// from Chapter 8/code/monsters/src/lib.rs:
fn print_from_monsters() {
    println!("Printing from crate monsters!");
}
```

Then compile the library with `cargo build`, producing a library file, `libmonsters.rlib`, in the `target/debug` folder.

Now we create a `main.rs` in the `src` folder, to make an executable that can call into our `monsters` library, and copy the original `main()` code from `traits.rs` into it, adding a call to `print_from_monsters()`:

```
// from Chapter 8/code/monsters/src/main.rs:
fn main() {
    print_from_monsters();
    let zmb1 = Zombie {health: 75, damage: 15};
    println!("Oh no, I hear: {}", zmb1.noise());
    zmb1.attack();
    println!("{:?}", zmb1);
}
```



This is a common design pattern: a library project containing an executable program that can be used to demonstrate or test the library.

`cargo build` will now compile both projects if there are no problems.

However, the code does not yet compile, and the compiler says `error: unresolved name` `` `print_from_monsters` ``; clearly the code for the function is not found.

The first thing we have to do is make the library code available to our main program, which is done with the following statement placed at the start:

```
extern crate monsters;
```

This statement will import all the (public) items contained in the monsters crate under a module with the same name. But this is not enough; we must also indicate that the `print_from_monsters` function is to be found in the module `monsters`. Indeed, the `monsters` crate creates an implicit module with the same name. So, we have to call our function as follows:

```
monsters::print_from_monsters();
```

Now we get `error: function `print_from_monsters` is private`, which tells us that the function is found, but is inaccessible in the library crate. This is easy; in the *Visibility of items* section we have seen how to remedy this: we prefix the function header with `pub`, like this:

```
pub fn print_from_monsters() { ... }
```

Now this part of our code works! Open a terminal, go (`cd`) to the `target/debug` folder, and start the `monsters` executable. This prints out the following:

```
Printing from crate monsters!
```

You will see that `extern crate abc` (with `abc` being a crate name) is often used in code, but you will never see `extern crate std`; why is this? The reason is that `std` is imported by default in every other crate. For the same reason, the contents of the `prelude` module are imported by default in every module.

Exporting a public interface

The next error the compiler throws at us is `error: Zombie does not name a structure`; clearly the code for the `Zombie` struct is not found. Because this struct also resides in the `monsters` module, the solution is easy: prefix `Zombie` with `monsters::`, like this:

```
let zmb1 = monsters::Zombie {health: 75, damage: 15};
```

Another `error: struct `Zombie` is private` makes it clear that we must mark the `Zombie` struct with `pub` like the following:

```
pub struct Zombie { ... }
```

Now we get an error on the line containing `zmb1.noise():error: type `monsters::Zombie` does not implement any method in scope named `noise``. The accompanying help note explains what to do and why:

```
help: methods from traits can only be called if the trait is in scope; the
following trait is implemented but not in scope, perhaps add a `use` for
it:
help: candidate #1: use `monsters::Monster`
```

So, let's add this to the code:

```
extern crate monsters;
use monsters::Monster;
```

The last error we have to solve occurs at the `use` line: `error: trait `Monster` is private – source trait is private`. Again, it is very logical; if we want to use a trait, it must be publicly visible:

```
pub trait Monster { ... }
```

Now `cargo build` is successful, and if we execute `monsters` the output is this:

```
Printing from crate monsters!
Oh no, I hear: Aaargh!
The Zombie bites! Your health lowers with 30 damage points.
Zombie { health: 75, damage: 15 }
```

This makes it clear that the things we want to make *visible* in our module (or put it another way, that we want to export) must be annotated with `pub`; they form the *interface* our module exposes to the outside world.

Adding external crates to a project

How should we use libraries written by others (choosing from the multitude of libraries available at `https://crates.io`) in our project? Cargo makes this very easy.

Suppose we want to use both the `log` and the `mac` library in the `monsters` project. `log` is a simple logging framework that gives us a number of macros such as `info!`, `warn!`, and `trace!` to log information messages. `mac` is a collection of useful macros, maintained by Jonathan Reem.

To get these libraries, we edit our `Cargo.toml` configuration file and add a `[dependencies]` section when it isn't already present. Beneath it, we specify the versions of the libraries we want to use:

```
[dependencies]
log = "0.3"
mac = "*"
```

A `*` denotes that any version is OK, and the most recent version will be installed.

If the library is not available at the `crates.io` repository, but only at GitHub, add a `[dependencies.libname]` section and refer to it with its complete URL for the Git property. For example, if your library depends on the `gtk` crate available at GitHub, add this to your `Cargo.toml`:

```
[dependencies.gtk]
git = "https://github.com/rust-gnome/gtk"
```

You can also refer to an `abc` library that still sits on your machine by including its path:

```
[dependencies.abc]
path = "path/to/abc"
```

The string value in the [dependencies] section (here `"0.3"`) follows the rules of semantic versioning, which are quite versatile (see `https://en.wikipedia.org/wiki/Software_versioning`); some examples include these:

```
1.* means all versions >= 1.0.0 but < 2.0.0
1.2.* means all versions >= 1.2.0 but < 1.3.0
= 1.2.3 means exactly that version
>= 1.2.3 means bigger than or equal to that version
>1.2.3 means bigger than that version
^1.2.3 means all versions starting from 1.2.3 but < 2.0.0
^0 means all versions starting from 0.0.0 but < 1.0.0
~1.2.3 means all versions starting from 1.2.3 but < 1.3.0
```

For other details, see `http://doc.crates.io/crates-io.html`.

Save the file and, in the `monsters` folder, issue this command:

```
cargo build
```

Cargo will take care of locally installing and compiling the libraries:



It will also automatically update the `Cargo.lock` file to register the installed versions of the libraries, so that subsequent project builds will always use the same versions (here `log v0.3.1` and `mac v0.0.1`). If you later want to update to the most recent version of a library, for example, for the `log` library, do a `cargo update -p log`, or a `cargo update` to update all libraries. This will download the latest crate versions for those crates indicated with version `*`. If you want a higher version for a crate, change its version number in `Cargo.toml`.

The `cargo check` command can also be useful; your project's code is checked by the compiler, but no executable code is generated.

Start using the libraries by importing their crates in the code:

```
#[macro_use]
extern crate log;
extern crate mac;
```

The `#[macro_use]` attribute allows the use of macros defined in the external crate (see the next section). Then we can, for example, use the `info!` macro from the `log` crate as follows:

```
info!("Gathering information from monster {:?}", zmb1);
```

Working with random numbers

Let's see another example of how to work with external crates using the `rand` crate.

Games often involve random choices, such as deciding which monster should attack our player or at which place you will find treasure; the `rand` ;crate provides for this. Start a Cargo project, `random`. To compile the `rand` crate with your program, add the following to the `Cargo.toml` configuration file:

```
[dependencies]
rand = "*"
```

To import the crate into your program, add the following lines at the start of your source code, `random\scr\main.rs`:

```
extern crate rand;
use rand::Rng;
```

The `rand` crate contains a `random()` function for generating a random value, but it can generate values of lots of different types, not only integer numbers; in other words, it is *generic*.

Using it looks like this: `let rnd = rand::random();`. This gives us the `error unable to infer enough type information about `_`; type annotations required`.

So we have to give Rust a hint regarding which type of value the `random()` function must generate. To tell it to produce integer values of 32 bits in size, attach `::<i32>` to the function name like this:

```
let rnd = rand::random::<i32>();
```

The following program illustrates some basic functionality for generating random values:

```
   // from Chapter 8/code/random/src/main.rs
fn main() {
    println!("Give me 5 random numbers:");
    for _ in 0..5 {
        let rnd = rand::random::<i32>();
        print!("{} / ", rnd);
    }
    println!("");
    println!("Give me 5 positive random numbers smaller than 32:");
    for _ in 0..5 {
        let rnd = (rand::random::<u32>() % 32) + 1;
        print!("{} / ", rnd);
    }
    println!("");

    let mut rng = rand::thread_rng();
    if rng.gen() { // random bool
        println!("i32: {}, u32: {}", rng.gen::<i32>(), rng.gen::<u32>())
    }

// generate a random number in a range, for example: 1 - 100:
    let secret_number = rand::thread_rng().gen_range(1, 101);
    println!("The secret number is: {}", secret_number);
```

```
    let tuple = rand::random::<(f64, char)>();
    println!("{:?}", tuple)
}
```

As always, build the crate with `cargo build`, and then execute it with `cargo run`.

This gives an output like this (because it is random, each time the program executes, the output will contain different values):

```
Give me 5 random numbers:
-1786096291 / -312872251 / 959357270 / 1391515785 / -1379700184 /
Give me 5 positive random numbers smaller than 32:
11 / 15 / 28 / 13 / 23 /
The secret number is: 11
(0.279622, '\u{583cf}')
"¼¢Q¢jê«;aù"
```

How would you make sure that only positive numbers are generated? You guessed it:

```
let rnd = rand::random::<u32>();
```

What if we need numbers between 1 and 32 (because, let us assume, there are 32 monsters lurking around in this level)? Use the `%` operator to return the remainder from the integer division by 32 (which goes from 0 to 32) and then add 1, like this:

```
let rnd = (rand::random::<u32>() % 32) + 1;
```

Macros

Macros are not new; we have already used them. Every time we called an expression that ended in an exclamation mark (`!`), we in fact called a **built-in macro**; the `!` sign distinguishes it from a function. In our code up until now, we have already used the `println!`, the `assert_eq!`, the `assert!`, the `panic!`, the `try!` and the `vec!` macros.

Why macros?

Macros make powerful language or syntax extensions, and thus *metaprogramming*, possible; for example, Rust has a `regex!` macro which allows for defining regular expressions in your program, which are compiled while your code is compiled. That way the regular expressions are verified and can be optimized at compile time, and so avoid runtime overhead.

Macros can capture repetitive or similar code patterns and replace them with other source code; the macro expands the original code into new code. This expansion happens early in compilation, before any static checking is done, so the resulting code is compiled together with the original code. In this sense, they much more resemble Lisp macros than C macros. Rust macros allow writing **Don't Repeat Yourself** ;(**DRY**) code, by factoring out the common parts of functions. But a macro is higher-level than a function, because a macro allows for generating the code for many functions at compile time.

So, as a Rust developer, you can also write your own macros, replacing repetitive code with much simpler code and thereby automating tasks. On the other side of the spectrum, this could even make it possible to write domain-specific languages. Macro coding follows a specific set of declarative pattern-based rules. Rust's macro system is also *hygienic*, which means no conflict is possible between variables used in the macro and variables outside the macro. Each macro expansion happens in a distinct syntax context, and each variable is tagged with the syntax context where it was introduced.

Macro code itself is harder to understand than normal Rust code, so it is not that easy to make. But on the other hand, you won't code macros every day; if a macro is tested, just use it. The full story of macro writing extends into advanced regions of Rust, but in the following sections we discuss the basic techniques for developing macros.

Developing macros

The basic structure of a macro definition for a macro with the name `mac1` is of this form:

```
macro_rules! mac1 {
(pattern) => (expansion);
(pattern) => (expansion);
...
}
```

So, we can see that the definition of a macro is also done through a macro, namely the macro `macro_rules!` ;As you can see, a macro is similar to a `match` block, defining one or more *rules* for pattern matching, each rule ending on a semicolon. Every rule consists of a pattern before the `=>` sign (also called a **matcher)** that is replaced with the expansion part during compilation, not while executing the code.

The macro can be called as either `mac1!()` or `mac1![]`; the braces contain the arguments.

The following macro, `welcome!`, expects no pattern and expands into a print statement by using the `println!` macro. It is trivial, but it demonstrates how macros work:

```
// from Chapter 8/code/macros.rs
macro_rules! welcome {
    () => (
        println!("Welcome to the Game!");
    )
}
```

You can invoke it by adding a bang sign, `!`, to its name:

```
fn main() {
   welcome!()
}
```

This prints out the following:

```
Welcome to the Game!
```

The pattern before => can contain an expression of the form `$arg:frag`:

- `$arg` binds a meta-variable, `arg`, to a value when the macro is called.
- Variables used inside a macro, such as `$arg`, are prefixed with a `$` sign, to distinguish them from normal variables in code.
- `frag` is a **fragment specifier**, and can be one of the following, with some examples added between `""`:
    - `expr`: `"42"` or `"2 + 2"` or `"if true then { 1 } else { 2 }"` or `"f(42)"`
    - `item`: `"fn average() { }"` or `"struct Person;"`
    - `block`: Is a sequence of instructions delimited by curly braces, such as `"{ log(error, "hi"); return 12; }"`
    - `stmt`: `"let x = 3"`
    - `pat`: `"Some(t)"` or `"(17, 'a')"` or `"_"`
    - `ty (type)`: `"i32"` or `"Vec<(char, String)>"` or `"&T"`
    - `ident`: `"x"` or `"42"`
    - `path`: `"T::SpecialA"`
    - `tt`: For more global elements



You can find more information on the meaning of these fragments in the official documentation ;`https://doc.rust-lang.org/stable/book/first-edition/macros.html`.

Any other Rust literals (tokens) that appear in a matcher must match exactly.

For example, consider the following macro, `mac1`:

```
macro_rules! mac1 {
    ($arg:expr) => (println!("arg is {}", $arg));
}
```

When you call `mac1!(42);` it will print out `arg is 42`.

`mac1` looks at its argument, `42`, as an expression (`expr`), and binds `arg` to that value.



Exercises:

- Write a macro, `mac2`, that triples its argument. Test it out for the following arguments: 5 and 2 + 3.
- Write a macro, `mac3`, that takes an identifier name and replaces it with a binding of that name to 42. (Hint: use `$arg:ident` instead of `$arg:expr`, `ident` is used for variable and function names.)
- Write a macro, `mac4`, that when invoked like `mac4!("Where am I?");` prints out `start - Where am I? - end`.
- Write a macro `says` that when invoked like ;`says!("Hi", "Jim");` prints out `He/She says: 'Hi' to Jim`.

(See the example code in `Chapter 8/exercises/macro_ex.rs`.)

Repetition

What if there is more than one argument? Then we can enclose the pattern with a `$(...)*`, where `*` means zero or more (instead of `*`, you can use `+` which means one or more).

For example, the following macro, `printall`, invokes `print!` on each of its arguments, which can be of arbitrary type and are separated by a `,`:

```
macro_rules! printall {
    ( $( $arg:expr ), * ) => ( {$( print!("{} / ", $arg) ); *} );
}
```

When called with `printall!("hello", 42, 3.14);` it will print out:

```
hello / 42 / 3.14 /
```

In the example, each argument (separated by commas) is substituted by a corresponding invocation of `print!` separated by a `/`. Note that on the right side we have to make a code block of the resulting print statements by enclosing it in `{ }`.

Creating a new function

Here is a macro, `create_fn`, to create a new function at compile time:

```
macro_rules! create_fn {
    ($fname:ident) => (
        fn $fname() {
        println!("Called the function {:?}()", stringify!($fname))
        }
    )
}
```

`stringify!` is a built-in macro that makes a string from its argument.

Now we can invoke this macro with `create_fn!(fn1);`. This statement does not sit inside `main()` or another function; it is transformed during compilation into the function definition. Then a normal call to that function `fn1()` will call it, here printing:

```
Called the function "fn1"().
```

Some other examples

In the following macro, `massert`, we mimic the behavior of the `assert!` macro, which does nothing when its expression argument is true, but panics when it is false:

```
macro_rules! massert {
    ($arg:expr) => (
            if $arg {}
            else { panic!("Assertion failed!"); }
    );
}
```

For example: `massert!(1 == 42);` will print out:

```
thread '<main>' panicked at 'Assertion failed!'
```

In the following statements, we test whether vector `v` contains certain elements:

```
let v = [10, 40, 30];
massert!(v.contains(&30));
massert!(!v.contains(&50));
```

The macro `unless` mimics an unless statement, where a `branch` is executed if the `arg` condition is not true. For example:

```
unless!(v.contains(&25), println!("v does not contain 25"));
```

It should print out:

```
v does not contain 25
```

Because the condition is not true. This is also a one-line macro:

```
macro_rules! unless {
    ($arg:expr, $branch:expr) => ( if !$arg { $branch }; );
}
```

The last example combines the techniques we have seen so far. In `Chapter 3`, *Using Functions and Control Structures*, in the *Attributes - testing* section, we saw how to make a test function with the `#[test]` attribute. Let us create a macro, `test_eq`, that when invoked with `test_eq!(seven_times_six_is_forty_two, 7 * 6, 42);` generates a test function:

```
#[test]
fn seven_times_six_is_forty_two() {
    assert_eq!(7 * 6, 42);
}
```

We also want a test that fails:

```
test_eq!(seven_times_six_is_not_forty_three, 7 * 6, 43);
```

The first argument to `test_eq` is the test's name, and the second and third arguments are values to be compared for equality, so in general `test_eq!(name, left, right);`.

Here, `name` is an identifier; `left` and `right` are expressions. Like the `create_fn` invocation previously, the `test_eq!` calls are written outside any function.

Now we can compose our macro as follows:

```
macro_rules! test_eq {
    ($name:ident, $left:expr, $right:expr) => {
        #[test]
```

```
            fn $name() {
                assert_eq!($left, $right);
            }
        }
    }
```

Create the test `runner` by calling `rustc --test macros.rs`.

When the `macros` executable is run, it prints out this:

```
running 2 tests
test seven_times_six_is_forty_two ... ok
test seven_times_six_is_not_forty_three ... FAILED
```

A macro can also be recursive, calling itself in the expansion branch. This is useful for processing tree-structured input, like when parsing HTML code.

The compiler can show us the expanded macro. Let us say we want to compile `main.rs` which contains some macro(s). Call `rustc` with the following options to show the expanded output (you will have to use the nightly compiler version here):

```
rustc main.rs --pretty=expanded -Z unstable-options
```

To debug a macro, and to give you more info about how your macro works, the macros `log_syntax` and `trace_macros(true)` can be used. Again, both macros are unstable features, so you must use them with a Rust compiler from the nightly branch and behind these feature gates:

```
#![feature(trace_macros)]
#![feature(log_syntax)]
```

For a complete example, see `macro_debug.rs`.

Using macros from crates

As we demonstrated at the end of the *Adding external crates to a project* section, loading all macros from an external crate is done by preceding the `extern crate abc` with the attribute `#[macro_use]`. If you only need the macros `mac1` and `mac2`, you can write this:

```
#[macro_use(mac1, mac2)]
extern crate abc;
```

If the attribute is not present, no macros are loaded from `abc`.

Moreover, inside the `abc` module, only macros defined with the `#[macro_export]` attribute can be loaded in another module.

To distinguish macros with the same name in different modules, use the `$crate` variable in the macro. Within the code of a macro imported from a crate, `abc`, the special macro variable `$crate` will expand to `::abc`.

Some other built-in macros

The `write!` macro allows you to write values into any kind of buffer, optionally using the same formatting capabilities as `format!` and `println!`. In the following example, we simply write a string to a vector:

```
// from Chapter 8/code/write.rs
// in order to be able to use write! on a vector
use std::io::Write;

fn main() {
    let mut vec1 = Vec::new();
    write!(&mut vec1, "test");
    println!("{:?}", vec1);
}
```

This prints out the following:

```
[116, 101, 115, 116].
```

A handy macro during development is `unimplemented!`, which you might have already seen in generated code. You can also use it yourself as a temporary placeholder for a function body you will write later:

```
// from Chapter 8/code/unimplemented.rs
   fn main() {
       unimplemented!();
   }
```

It will compile just fine, but when executed it will panic:

```
thread 'main' panicked at 'not yet implemented'
```

This indicates the code line where it found the macro. That way you won't forget to write the code!

A similar macro is called `unreachable!` Put this in a code location that you think will never be executed, like in the following if statement:

```
// from Chapter 8/code/unreachable.rs
fn main() {
    if false {
        unreachable!();
    }
    unreachable!();
}
```

But if it does execute, it gives you a panic to indicate that code line was nevertheless reached:

```
thread 'main' panicked at 'internal error: entered unreachable code'.
```

Summary

In this chapter, we learned how to structure modules into crates to make our code more flexible and modular. We now also know the basic rules for writing macros in order to write more compact and less repetitive code.

In the following chapter, we will explore the power of Rust when it comes to concurrent and parallel execution of code, and how Rust also preserves memory safety in this area.

9
Concurrency - Coding for Multicore Execution

As a modern system-level programming language, Rust has to have a good method for executing code concurrently and in parallel on many processors simultaneously. And indeed, it does: Rust provides a wide selection of concurrency and parallel tools. Its type system is strong enough to write concurrency primitives that have properties unlike anything that has existed before. In particular, it can encode a wide selection of memory safe parallel abstractions that are also guaranteed to be data-race free, and no garbage collector is used. This is mind-blowing; no other language can do this. All these features are not ingrained in the language itself, but provided by libraries so that improved or new versions can always be built. Developers should choose the tool that is right for the job at hand, or that can improve on or develop new tools.

We will discuss the following topics:

- Concurrency and threads
- Shared mutable states
- Communication through channels

Concurrency and threads

A system is concurrent when several computations are executing at the same time and potentially interacting with each other. The computations can only run in parallel (that is, simultaneously) when they are executing on different cores or processors.

An executing Rust program consists of a collection of native **Operating System** (**OS**) threads; the OS is also responsible for their scheduling. The unit of computation in Rust is called a `thread`, which is a type defined in the `std::thread` module. Each thread has its own stack and local state.

Until now, our Rust programs only had one thread, the `main` thread, corresponding with the execution of the `main()` function. But a Rust program can create lots of threads to work simultaneously when needed. Each `thread` (not only `main()`) can act as a parent and generate any number of children threads.

Data can either be:

- Shared across threads (see the shared mutable states through atomic types section)
- Sent between threads (see the communication through channels section)

Creating threads

A `thread` can be created by spawning it; this creates an independent detached child thread that in general can outlive its parent. This is demonstrated in the following code snippet:

```
// code from Chapter 9/code/thread_spawn.rs:
use std::thread;
use std::time;

fn main() {
    thread::spawn(move || {
 println!("Hello from the goblin in the spawned thread!");
    });
}
```

The argument to `spawn` is a closure (here, without parameters, it is indicated as `||`), which is scheduled to execute independently from the parent thread, which is shown here as `main()`. Notice that it is a moving closure which takes ownership of the variables in a given context. Our closure here is a simple print statement, but in a real situation this could be replaced by a heavy or time-consuming operation.

When we execute this code, we normally don't see any output. Why is this? It turns out that `main()` is a bad parent (as far as threading is concerned) and doesn't wait for its children to end properly. The output of the spawned thread only becomes visible if we let `main()` pause for a brief moment before terminating. This can be done with the `thread::sleep` method,which takes the `Duration`, a type defined in the `std::time` module:

```
    fn main() {
     thread::spawn(move || { ...  });
     thread::sleep(time::Duration::from_millis(50));
}
```

This prints out the following:

```
Hello from the goblin in the spawned thread!.
```

In general, this pause period is not needed. Child threads that are spawned can live longer than their parent threads, and will continue to execute when their parents have already stopped.

Better practice in this case, however, is to capture the `handle` join that `spawn` returns in a variable. Calling the `join()` method on `handle` will block the parent thread, making it wait until the child thread is finished. This returns a `Result` instance. `unwrap()` will take the value from `Ok`, returning the result of the child thread, which is `()` in this case because it is a print statement, or panic in the `Err` case:

```
fn main() {
    let handle = thread::spawn(move || {
        println!("Hello from the goblin in the spawned thread!");
    });
// do other work in the meantime
   let output = handle.join().unwrap(); // wait for child thread
   println!("{:?}", output); // ()
}
```

If no other work has to be done while the child thread is executing, we can also write this:

```
thread::spawn(move || {
// work done in child thread
}).join();
```

In this case, we are waiting synchronously for the child thread to finish, so there is no good reason to start a new thread.

Why do we need a moving closure when spawning a thread? Examine the following code, which does not compile:

```
// code from Chapter 9/code/moving_closure.rs:
use std::thread;

fn main() {
      read();
}

fn read() {
    let book = read_book("book1.txt");
    thread::spawn(|| {
        println!("{:?}", book);
    });
}
fn read_book(s: &str) { }
```

This gives us `error: closure may outlive the current function, but it borrows `book`, which is owned by the current function`. Indeed, the closure may be run by the spawned thread long after the `read()` function has terminated, as it has freed the memory of `book`. But if we make it a moving closure, which takes ownership of its environment, the code compiles:

```
thread::spawn(move|| {
        println!("{:?}", book);
   });
```

Setting the thread's stack size

The `RUST_MIN_STACK` environment variable can override the default stack size of 2 MB for created threads, but not for the main thread. You can also set the thread's stack size in code by using a `thread::Builder`, like this:

```
let child = thread::Builder::new().stack_size(32 * 1024 * 1024).spawn(move
|| {
   // code to be executed in thread
}).unwrap();
```

Here, the child thread gets 32 MB of stack space.

Starting a number of threads

Each thread has its own stack and local state, and by default no data is shared between threads unless it is immutable data. Generating threads is a very lightweight process; for example, starting tens of thousands of threads only takes a few seconds. The following program does just that, and prints out the numbers 0 to 9,999:

```
// code from Chapter 9/code/many_threads.rs:
use std::thread;

static NTHREADS: i32 = 10000;

fn main() {
println!("************************* Before the start of the threads");
for i in 0..NTHREADS {
         let _ = thread::spawn(move || {
              println!("this is thread number {}", i)
         });
      }
thread::sleep(time::Duration::from_millis(500));
println!("************************* All threads finished!");
}
```

Because the numbers are printed in independent threads, the order is not preserved in the output, so it could start with this:

```
************************* Before the start of the threads
this is thread number 1
this is thread number 3
this is thread number 4
this is thread number 2
this is thread number 6
this is thread number 5
this is thread number 0
...
************************* All threads finished!
```

Notice we need to add a `sleep()` method to ensure all spawned threads get finished, otherwise we get error messages such as this:

```
thread '<unnamed>' panicked at 'cannot access stdout during shutdown'
```

A question that often arises is how many threads do I have to spawn? The basic rule is, for CPU intensive tasks, have the same number of threads as CPU cores.

This number can be retrieved in Rust by using the `num_cpus` crate.

Let's make a new project with `cargo new many_threads --bin`:

1. Add the crate dependency to `Cargo.toml`:

```
[dependencies]
num_cpus = "*"
```

2. Then change `main.rs` to this:

```
extern crate num_cpus;

fn main() {
   let ncpus = num_cpus::get();
    println!("The number of cpus in this machine is: {}", ncpus);
}
```

3. From within the `many_threads` folder, do a `cargo build` to install the crate and compile the code. Executing the program with `cargo run` gives us the following output (dependent on the computer):

```
The number of cpus in this machine is: 8
```

For efficiency, it is best to start with this, or any other number of threads in a pool. This functionality is provided by the `threadpool` crate, which we can get by adding the `threadpool = "*"` dependency to `Cargo.toml` and doing a `cargo build`.

Add the following code to the start of the file:

```
extern crate threadpool;

use std::thread;
use std::time;
 use threadpool::ThreadPool;
Add this code to the main() function:
let pool = ThreadPool::new(ncpus);

for i in 0..ncpus {
      pool.execute(move || {
         println!("this is thread number {}", i)
   });
}
   thread::sleep(time::Duration::from_millis(50));
```

When executed, this yields the following output:

```
this is thread number 0
this is thread number 5
this is thread number 7
this is thread number 3
this is thread number 4
this is thread number 1
this is thread number 6
this is thread number 2
```

A thread pool is used for running a number of jobs on a fixed set of parallel worker threads; it creates the given number of worker threads and replenishes the pool if any thread panics.

Panicking threads

What happens when one of the spawned threads gets into a panic? No problem, the threads are isolated from each other, and only the panicking thread will crash after having freed its resources; the parent thread is not affected. In fact, the parent can test the `is_err` return value from `spawn`, like this:

```
// code from Chapter 9/code/panic_thread.rs:
use std::thread;

fn main() {
   let result = thread::spawn(move || {
      panic!("I have fallen into an unrecoverable trap!");
   }).join();

   if result.is_err() {
      println!("This child has panicked");
   }
}
```

This prints out the following:

```
thread '<unnamed>' panicked at 'I have fallen into an unrecoverable trap!'
This child has panicked
```

To put it another way, the thread is a unit of failure isolation.

Thread safety

Traditional programming with threads is very difficult to get right if you allow the different threads to work on the same mutable data, or so-called **shared memory**. When two or more threads simultaneously change data, then data corruption (also called **data racing**) can occur due to the unpredictability of the thread's scheduling.

In general, data (or a type) is said to be **thread-safe** when its contents will not be corrupted by the execution of different threads. Other languages offer no such help, but the Rust compiler simply forbids non-thread-safe situations to occur. The same ownership strategy that we looked at to allow Rust to prevent memory safety errors also enables you to write safe, concurrent programs.

Consider the following program:

```
// code from Chapter 9/code/not_shared.rs:
use std::thread;
use std::time;

fn main() {
    let mut health = 12;
    for i in 2..5 {
        thread::spawn(move || {
            health *= i;
        });
    }
 thread::sleep(time::Duration::from_secs(2));
    println!("{}", health); // 12
}
```

Our initial health is 12, but there are three fairies that can double, triple, and quadruple our health. We let each of them do that in a different thread, and after the threads are finished we expect a `health` of 288 (= *12 * 2 * 3 * 4*). But after their magic actions, our `health` is still `12`, even if we wait long enough to be sure that the threads are finished. Clearly, the three threads worked on a copy of our variable, not on the variable itself. Rust does not allow the `health` variable to be shared among the threads to prevent data corruption. In the next section, we will explore how we can use mutable variables that are shared between threads.

Shared mutable states

How can we make the `not_shared.rs` program give us the correct result? Rust provides tools, such as **atomic types** from the submodule `std::sync::atomic`, to handle shared mutable state safely. In order to share data, you need to wrap the data in some of these sync primitives, such as `Arc`, `Mutex`, `RwLock`, `AtomicUSize`, and so on.

Basically, the principle of locking is used, as in operating systems and database systems. Exclusive access to a resource is given to the thread that has obtained a **lock** (also called a **mutex**, from mutually exclusive) on the resource. A lock can only be obtained by one thread at a time. In this way, two threads cannot change this resource at the same time, so no data races can occur; locking atomicity is enforced when required. When the thread that has acquired the lock has done its work, the lock is removed and another thread can then work with the data.

In Rust, this is done with the generic `Mutex<T>` type from the `std::sync` module. The sync comes from synchronize, which is exactly what we want to do with our threads.

The `Mutex` ensures that only one thread at a time can change the contents of our data.

We must make an instance of this type, wrapping our data like this:

```
// code from Chapter 9/code/thread_safe.rs:
use std::sync::{Arc, Mutex};
 let data = Mutex::new(health);
Now, within the for loop immediately after that, we spawn the new thread,
and we place a lock on the health object:
     for i in 2..5 {
thread::spawn(move || {
             let mut health = data.lock().unwrap();
             // do other things
         }
     }
```

The call to `lock()` will return a reference to the value inside the `Mutex`, and it will block any other calls to `lock()` until that reference goes out of scope, which is at the end of the thread closure. Then the thread has done its work and the lock is automatically removed. But for now, we will still get an `error: capture of moved value: 'data'`. This is because the `data` variable cannot be moved to another thread multiple times.

This problem can be solved by using an equivalent of the `Rc` pointer from `Chapter 7`, *Ensuring Memory Safety and Pointers*, in the Reference counting section. Indeed, the situation is very similar: all the threads need a reference to the same data, our `health` variable. So we apply the same techniques from `Chapter 6`, *;Using Traits and OOP in Rust* here: make an `Rc` pointer to point to our data, and make a `clone()` of the pointer for each reference that is needed. But a simple `Rc` pointer is not thread-safe, therefore we need a special version of it that is thread-safe, the so-called **atomic reference counted pointer**, or `Arc<T>`. Atomic means it is safe across threads, and it is also generic.

So, we envelop our `health` variable inside an `Arc` pointer, like this:

```
    let data = Arc::new(Mutex::new(health));
```

And in the `for` loop, we make a new pointer to the `Mutex` with `clone`:

```
    for i in 2..5 {
        let mutex = data.clone();
        thread::spawn(move || {
                  let mut health = mutex.lock().unwrap();
                *health *= i;
});
    }
```

Each thread now works with a copy of the pointer obtained by `clone()`. The `Arc` instance will keep track of the number of references to `health`. A call to `clone()` will increment the reference count on `health`. The `mutex` reference goes out of scope at the end of the thread closure, which will decrement the reference count. `Arc` will free the associated `health` resource when that reference count is zero.

Calling `lock()` gives the active thread exclusive access to the data. In principle, acquiring the lock might fail, so it returns a `Result<T, E>` object. In the preceding code, we assume everything is OK. `unwrap()` is a quick means to return a reference to the data, but in a failure case, it panics.

Quite a few steps were involved here, so we will repeat the code in its entirety here, this time providing robust error handling to replace `unwrap()`. Digest each line with the explanations given previously:

```
// code from Chapter 9/code/thread_safe.rs:
use std::thread;
use std::sync::{Arc, Mutex};

fn main() {
   let mut health = 12;
   println!("health before: {:?}", health);
```

```
    let data = Arc::new(Mutex::new(health));
    for i in 2..5 {
         let mutex = data.clone();
         thread::spawn(move || {
             let health = mutex.lock();
             match health {
                 // health is multiplied by i:
Ok(mut health) => *health *= i,
                 Err(str) => println!("{}", str)
             }
         }).join().unwrap();
     };
     health = *data.lock().unwrap();
     println!("health after: {:?}", health);
}
```

This prints out the following:

```
health before: 12
health after: 288
```

(Indeed, *288 = 12 * 2 * 3 * 4*)

We `join()` the threads to give them time to do their work; `data` is a reference, so we need to dereference it to obtain the `health` value:

```
health = *data.lock().unwrap();
```

The mechanism outlined previously, using a combined `Mutex` and `Arc`, is advisable when the shared data occupies a significant amount of memory, because with an `Arc`, the data will no longer be copied for each thread. The `Arc` acts as a reference to the shared data and only this reference is shared and cloned.

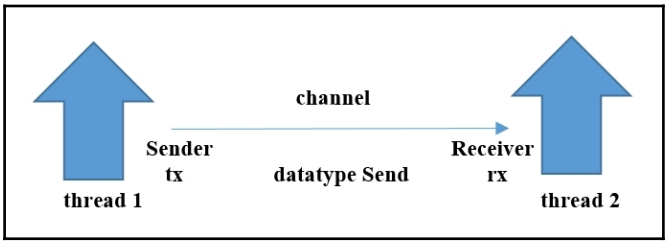
The Sync trait

An `Arc<T>` object implements the `Sync` trait (while `Rc` does not), which indicates to the compiler that it is safe to use concurrently with multiple threads. Any data that has to be shared simultaneously among threads must implement the `Sync` trait. The ;`T` ;type is `Sync` if there is no possibility of data races when passing `&T` references between threads; in short, `&T` ;is thread-safe. All simple types, such as integers and floating point numbers, are `Sync`, as well as all composite types (such as structs, enums, and tuples) that are built with simple types. Any types that only contain things that implement `Sync` are automatically `Sync`.

Communication through channels

Data can also be exchanged between threads by passing messages among them. This is implemented in Rust by channels, which are like unidirectional pipes that connect two threads; data is processed first-in, first-out.

Data flows over this channel between two endpoints: from the `Sender<T>` to the `Receiver<T>`, both of which are generic and take the type `T` of the message to be transferred (which obviously must be the same for the `Sender` and `Receiver`). In this mechanism, a copy of the data to share is made for the receiving thread, so you wouldn't want to use this for very large data:



To create a channel, we need to import the `mpsc` submodule from `std::sync (mpsc`, which stands for multi-producer, single-consumer communication primitives, and then use the `channel()` method:

```
// code from Chapter 9/code/channels.rs:
use std::thread;
use std::sync::mpsc::channel;use std::sync::mpsc::{Sender, Receiver};
fn main() {
   let (tx, rx): (Sender<i32>, Receiver<i32>) = channel();
}
```

This creates a tuple of endpoints, `tx` (t stands for transmission) being the `Sender` and `rx` (r stands for receiver) being the `Receiver`. We have indicated that we will send `i32` integers over the channel, but the type annotations are not needed if the compiler can deduce the channel datatype from the rest of the code.

Sending and receiving data

Which datatypes can be sent over a channel? Rust requires that data to be sent over a channel must implement the `Send` trait, which guarantees that ownership is transferred safely between threads. Data that does not implement `Send` cannot leave the current thread. An `i32` is `Send` because we can make a copy, so let's do that in the following code snippet:

```
fn main() {
   let (tx, rx) = channel();
   thread::spawn(move|| {
     tx.send(10).unwrap();
   });
   let res = rx.recv().unwrap();
   println!("{:?}", res);
}
```

This of course prints `10`.

Here, `tx` is moved inside the closure. A better way to write `tx.send(10).unwrap()` is the following:

```
tx.send(10).ok().expect("Unable to send message");
```

This will ensure that, in the case of a problem, a message is reported.

`send()` is executed by the child thread; it queues a message (a data value, here 10) in the channel and does not block it. The `recv()` is done by the parent thread; it picks a message from the channel and blocks the current thread if there are no messages available (if you need to do this in a non-blocking fashion, use `try_recv()`).

If you do not need to process the received value, the block can be written as follows:

```
let _ = rx.recv();
```

The `send()` and `recv()` operations return a `Result`, which can be of type `Ok(value)` or an error, `Err`. Full error handling is omitted here, because in the case of an `Err` the channel does not work anymore, and it is better for that thread to fail (panic) and stop.

In a general scenario, we could make a child thread execute a long computation, and then receive the result in the parent thread, like this:

```
// code from Chapter 9/code/channels2.rs:
use std::thread;
use std::sync::mpsc::channel;

fn main() {
```

```
    let (tx, rx) = channel();
    thread::spawn(move|| {
         let result = some_expensive_computation();
         tx.send(result).ok().expect("Unable to send message");
     });

     some_other_expensive_computation();
     let result = rx.recv();
     println!("{:?}", result);
}

fn some_expensive_computation() -> i32 { 1 }
fn some_other_expensive_computation() { }
```

`result` here has the value `Ok(1)`.

Making a channel

An elegant code pattern is shown in the following code snippet, where the channel is created in the `make_chan()` function, which returns the receiving endpoint for the calling code:

```
// code from Chapter 9/code/make_channel.rs:
use std::sync::mpsc::channel;
use std::sync::mpsc::Receiver;
fn make_chan() -> Receiver<i32> {    let (tx, rx) = channel();
tx.send(7).unwrap();    rx}
fn main() {
    let rx = make_chan();
    if let Some(msg) = rx.recv().ok() {
      println!("received message {}", msg);
    };
}
```

This prints out the following:

```
received message 7
```

Sending struct values over a channel

The following example shows how to use channels to send struct values:

```
// see code from Chapter 9/code/channel_struct.rs
use std::thread;
use std::sync::mpsc::channel;

struct Block {
    value: i32
}

fn main() {
    let (tx1, rx1) = channel::<Block>();
    let (tx2, rx2) = channel::<Block>();
    thread::spawn(move|| {
        let mut block = rx1.recv().unwrap();
        println!("Input: {:?}", block.value);
        block.value += 1;
        tx2.send(block).unwrap();
    });
    let input = Block{ value: 1 };
    tx1.send(input).unwrap();
    let output = rx2.recv().unwrap();
    println!("Output: {:?}", output.value);
}
```

This produces the following output:

```
Input: 1
Output: 2
```

Sending references over a channel

Even references and pointers can be sent over a channel, like in this example:

```
// see code from Chapter 9/code/channel_box.rs
use std::thread;
use std::sync::mpsc;

trait Message : Send {
    fn print(&self);
}

struct Msg1 {
    value: i32
}
```

```
impl Message for Msg1 {
    fn print(&self) {
        println!("value: {:?}", self.value);
    }
}

fn main() {
    let (tx, rx) = mpsc::channel::<Box<Message>>();
    let handle = thread::spawn(move|| {
        let msg = rx.recv().unwrap();
        msg.print();
    });
    let msg = Box::new(Msg1{ value:1 });
    tx.send(msg).unwrap();
    handle.join().ok();
}
```

This prints out the following:

```
value: 1
```



Exercise:
Make a program, `shared_channel.rs`, that lets any number of threads share a channel to send in a value, and one receiver that collects all the values.
(Hint: use `clone()` to give each thread access to the sending endpoint `tx`. See the example code in `Chapter 9/exercises/shared_channel.rs`.)

Synchronous and asynchronous

The kind of sending channel we have used up until now has been asynchronous, which means it does not block the executing code. Rust also has a synchronous channel type, `sync_channel`, where the `send()` ;method blocks the code execution if its internal buffer becomes full; it waits until the parent thread starts receiving the data. In the following code, this type of channel is used to send a value of struct `Msg` over the channel:

```
// code from Chapter 9/code/sync_channel.rs:
use std::sync::mpsc::sync_channel;
use std::thread;

type TokenType = i32;

struct Msg {
    typ: TokenType,
    val: String,
```

```
}

fn main() {
    let (tx, rx) = sync_channel(1); // buffer size 1
 tx.send(Msg {typ: 42, val: "Rust is cool".to_string()}).unwrap();
    println!("message 1 is sent");
    thread::spawn(move|| {
 tx.send(Msg {typ: 43, val: "Rust is still cool".to_string()}).unwrap();
        println!("message 2 is sent");
    });

    println!("Waiting for 3 seconds ...");
     thread::sleep(time::Duration::from_secs(3));

    if let Some(msg) = rx.recv().ok() {
println!("received message of type {} and val {}", msg.typ, msg.val);
    };
    if let Some(msg) = rx.recv().ok() {
println!("received second message of type {} and val {}", msg.typ,
msg.val);
    };
}
```

This prints the following:

```
message 1 is sent
Waiting for 3 seconds
```

Then, after 3 seconds, it prints the following:

```
message 2 is sent
received message of type 42 and val Rust is cool
received second message of type 43 and val Rust is still cool
```

This clearly shows that the second message can only be sent when the buffer is emptied by receiving the first message.



Question:

Explain what happens when the second message is also sent from within the main thread, not a separate thread.

Summary

In this chapter, we explored Rust's lightweight thread processes, how to create them, how to let them share data, and how to let them communicate data through channels.

In the following chapter, we will look at the boundaries. We will examine what we have to do in Rust when we go so low-level that the compiler cannot guarantee safety anymore, and how to interface with other languages such as C.

10
Programming at the Boundaries

In this chapter, we will look at situations where we have to leave safety boundaries, such as when interfacing with C programs. Rust minimizes potential dangers when doing this.

We will discuss the following topics:

- When is code unsafe
- Raw pointers
- Interfacing with C
- Inlining assembly code
- Calling Rust from other languages

When is code unsafe

There are situations in which even the Rust compiler cannot guarantee that our code will behave in a safe manner. This could occur when:

- We need to program against the metal, close to the operating system, processors, and hardware
- We want to work with the same amount of control as is possible in C
- We delegate a part of the program execution to an unsafe language, such as C
- We want to use inline assembly language

Rust allows us to do all of these, but we have to envelop this possibly dangerous code in an `unsafe` block:

```
unsafe {
  // possibly dangerous code
}
```

This means that the programmer takes full responsibility for the code in the block. The unsafe block is a promise to the compiler that the danger does not leak out of the block. The compiler will check the code areas marked as `unsafe` more loosely and allow otherwise forbidden manipulations, but a number of rules in the ownership system (see `Chapter 7`, *Ensuring Memory Safety and Pointers*) still remain in place.

The clear advantage here is that problem areas now appear very well isolated; if a problem occurs, we know that it can only occur in the code areas that are marked as unsafe. Having a code base where 99 percent of the code is safe and 1 percent is unsafe is much easier to maintain than a code base with 100 percent unsafe code, as in C!

So, what can we do in an `unsafe` block?

- Work with *raw pointers*, especially dereferencing (see the *Raw pointers* section).
- Call a function in another language through an **Foreign Function Interface** (**FFI**), see the *Interfacing with C* section.
- Use assembly code in our Rust code.
- Use `std::mem::transmute` to convert simple types, bitwise. Here is an example of its use, transforming a string to a slice of bytes:

```
// code from Chapter 10/code/unsafe.rs:
use std::mem;

fn main() {
   let v: &[u8] = unsafe {
      mem::transmute("Gandalf")
   };
   println!("{:?}", v);
}
```

- This prints out `[71, 97, 110, 100, 97, 108, 102]`.
- Use (read or change) mutable static variables. This can only be done in an unsafe block; it is a very bad idea anyway. Here is an example:

```
   // code from Chapter 10/code/unsafe.rs:

static mut N: i32 =  42;
fn main() {
// N = 108; // use of mutable static requires unsafe         // function or
block
    unsafe {
       println!("{:?}", N );  // 42
       N = 108;
       println!("{:?}", N );  // 108
```

```
    }
}
```

An `unsafe` block can also call `unsafe` functions that perform these dangerous operations, marked like this: `unsafe fn dangerous() { }`.

Using std::mem

In `unsafe` code, the use of the modules `std::mem` (which contains functions to work with memory at a low level) and `std::ptr` (which contains functions to work with raw pointers) is common, as we saw with `std::mem::transmute`.

Here are some more examples.

To swap two variables by explicitly using pointers, use `std::mem::swap` like this:

```
   // code from Chapter 10/code/swap.rs:
use std::mem;

fn main() {
   let mut n = 0;
   let mut m = 1;
   mem::swap(&mut n, &mut m);
   println!("n: {} m: {}", n, m);
}
```

This prints out the following:

```
n: 1 m: 0
```

As another example, the `mem::size_of_val()` function from the `mem` module takes a reference to a value and returns the number of bytes it occupies in memory. `mem::size_` of returns the size of the given type in bytes as a `u8`. For an example of its use, see the following code:

```
// code from Chapter 10/code/size_of_val.rs:                          use
std::mem;
fn main() {
    let arr = ["Rust", "Go", "Swift"];
    println!("array arr occupies {} bytes", mem::size_of_val(&arr));
println!("The size of an isize: {} bytes", mem::size_of::<isize>());
}
```

This prints out the following:

```
array arr occupies 48 bytes
The size of an isize: 8 bytes
```

Raw pointers

In unsafe code blocks, Rust allows the use of a new kind of pointers called **raw pointers**. These pointers have no built-in security, but you can work with them with the same freedom as you would with C pointers. They are written as:

- `*const T` for a pointer to an immutable value, or type `T`
- `*mut T` for a mutable pointer

These can point to invalid memory, and the memory resources need to be manually freed. This means that a raw pointer could inadvertently be used after freeing the memory it points to. Also, multiple concurrent threads have non-exclusive access to mutable raw pointers. Because you're not sure of its contents (at least, we have no compiler guarantee of valid content), dereferencing a raw pointer can also lead to program failure.

This is why dereferencing a raw pointer can only be done inside an unsafe block, as illustrated in the following code fragment:

```
// code from Chapter 10/code/raw_pointers.rs:
let p_raw: *const u32 = &10;
// let n = *p_raw; // compiler error!
unsafe {
     let n = *p_raw;
      println!("{}", n); // prints 10
}
```

If you try to do this in normal code, you get the error:

```
 error: dereference of unsafe pointer requires unsafe function or block
[E0133]
```

We can make raw pointers safe outside of references, implicitly or explicitly, with `& as *const`, as in the following snippet:

```
let gr: f32 = 1.618;
let p_imm: *const f32 = &gr as *const f32; // explicit cast
let mut m: f32 = 3.14;
let p_mut: *mut f32 = &mut m; // implicit cast
```

However, converting a raw pointer to a reference, which should be done through the `&*` (address of a dereference) operation, must be done within an unsafe block:

```
unsafe {
     let ref_imm: &f32 = &*p_imm;     let ref_mut: &mut f32 = &mut *p_mut;
   }
```



It is recommended to abundantly use `assert!` statements inside unsafe code, to check at runtime that it is doing what you expect. For instance, before dereferencing a raw pointer `ptr` of unknown origin, always call `assert!(!ptr.is_null());` to ensure that the pointer points to a valid memory location.

Raw pointers also allow pointer arithmetic with the `offset` method, but only in unsafe blocks. For the example code, see `pointer_offset.rs`:

```
fn main() {
    let items = [1u32, 2, 3, 4];
    let ptr = &items[1] as *const u32;
    println!("{}", unsafe { *ptr });
    println!("{}", unsafe { *ptr.offset(-1) });
    println!("{}", unsafe { *ptr.offset(1) });
}
```

However, doing arithmetic with pointers, as in C, is not allowed:

```
    println!("{}", unsafe { ptr + 2});
```

This gives us the following compiler error:

```
 error: binary operation `+` cannot be applied to type `*const u32` [E0369]
```

Raw pointers can also be useful when defining other, more intelligent pointers. For example, they are used to implement the `Rc` and `Arc` pointer types.

Interfacing with C

Because of the vast functionality that exists in C code, it can sometimes be useful to delegate processing to a C routine, instead of writing everything in Rust.

You can call all functions and types from the C standard library by using the `libc` crate, which must be obtained through Cargo.

You have to add the following to `Cargo.toml`:

```
[dependencies]
   libc = "*"
```

If you use Rust's nightly version, this is not necessary, but you have to use a feature attribute (or a feature gate, as they are also called) at the start of your code:

```
#![feature(libc)]
```



Feature gates are common in Rust to enable the use of a certain functionality, but they are not available in stable Rust, only in the current development branch (nightly release).

If you have both the stable and nightly versions installed, you can easily switch between them using the commands `rustup default nightly` and `rustup default stable`.

Then, simply add the following to your Rust code:

```
extern crate libc;
```

To import C functions and types, you can sum them up like this:

```
use libc::{c_void, size_t, malloc, free};
```

You can also use the wildcard `*` to make them all available, like in `use libc::*;`.

To work with C (or another language) from Rust, you will have to use the **Foreign Function Interface** (**FFI**); its utilities are in the `std::ffi` module. A foreign function interface is a mechanism by which a program written in one programming language can call routines or make use of services written in another language. But do realize that FFI calls cause significant overhead, so use them only when it is worth it.

Here is a simple example of calling C to print out a Rust `String` with the C function `puts`:

```
// code from Chapter 10/code/calling_libc.rs:
extern crate libc;
use libc::puts;
use std::ffi::CString;

fn main() {
   let sentence = "Merlin is the greatest magician!";
   let to_print = CString::new(sentence).unwrap();
   unsafe {
       puts(to_print.as_ptr());    }
   }
```

This prints out the sentence:

```
Merlin is the greatest magician!
```

The `new()` method of `CString` will produce a C-compatible string (ending with 0 bytes) from the Rust `String`. The `as_ptr()` method returns a pointer to this C string.

Using a C library

Suppose we want to calculate the tangents of a complex number. The `num` crate offers this functionality; however, for the purposes of this exercise, we will call the `ctanf` function from the C library `libm` instead. This is a collection of mathematical functions implemented in C. The following code defines a complex number as a simple struct.



Warning: This code only works on OS X and Linux, not on Windows.

```
// code from Chapter 10/code/calling_clibrary.rs:
#[repr(C)]
#[derive(Copy, Clone)]
#[derive(Debug)]
struct Complex {
    re: f32,
    im: f32,
}

#[link(name = "m")]extern {   fn ctanf(z: Complex) -> Complex;}

fn tan(z: Complex) -> Complex {
    unsafe { ctanf(z) }
}

fn main() {
    let z = Complex { re: -1., im: 1. }; // z is -1 + i
    let z_tan = tan(z);
    println!("the tangents of {:?} is {:?}", z, z_tan);
}
```

This program prints out the following:

```
the tangents of Complex { re: -1, im: 1 } is Complex { re: -0.271753, im:
1.083923 }
```

The `#[derive(Debug)]` attribute is necessary because we want to show the number in a `{:?}` format string. `#[derive(Copy, Clone)]` is needed because we want to use `z` in the `println!` statement, after we have moved it by calling `tan(z)`.The function of `#[repr(C)]` is to reassure the compiler that the type we are passing to C is foreign-function safe, and it tells `rustc` to create a struct with the same layout as C.

The signatures of the C functions we want to use must be listed in an `extern{}` block. The compiler cannot check these signatures, so it is important to specify them accurately to make the correct bindings at runtime. This block can also declare global variables exported by C to be used in Rust. These must be marked with `static` or `static mut`, for example, `static mut version: libc::c_int`.

The `extern` block must be preceded by a `#[link(name = "m")]` attribute to link in the `libm` library. This instructs `rustc` to link to that native library so that symbols from that library are resolved.

The C call itself must evidently be done inside an `unsafe {}` block. This block is enveloped inside the `tan(z)` wrapper function, which only uses Rust types. The wrapper can be exposed as a safe interface, hiding the unsafe calls and type conversions between Rust and C types, especially C pointers. When the C code returns a resource, the Rust code must contain destructors for these values to ensure their memory is released.

Inlining assembly code

In Rust, we can embed assembly code. This should be extremely rare, but one can imagine situations where this might be useful, for example, when you require utmost performance or very low-level control. But of course, the portability of your code and perhaps its stability decrease when doing this. The Rust compiler will probably generate better assembly code than you would write, so it isn't worth the effort most of the time.

The mechanism works by using the `asm!` macro, as in this example, where we calculate `a - b` in the `subtract` function by calling the assembly code:

```
// code from Chapter 10/code/asm.rs:
#![feature(asm)]

fn subtract(a: i32, b: i32) -> i32 {
    let sub: i32;
    unsafe {
        asm!("sub $2, $1; mov $1, $0" : "=r"(sub) : "r"(a), "r"(b)
);
    }
```

```
        sub
    }

    fn main() {
        println!("{}", subtract(42, 7))
```

This prints out the result **35**.

`asm!` can only be used with the feature gate `#![feature(asm)]`.

`asm!` has a number of parameters separated by `:`. The first is the assembly template, containing the assembly code as a string, then come the output and input operands.

You can indicate the kind of processors your assembly code is meant to execute on with the `cfg` attribute and its value, `target_arch`, for example:

```
#[cfg(any(target_arch = "x86", target_arch = "x86_64"))]
```

The compiler will then check whether you have specified valid assembly code for that processor.



This feature is not yet enabled in Rust 1.21 on the stable release channel. To use this mechanism (or other unstable features) in the meantime, you have to use Rust from the master branch (which is the nightly release). Install that version and change to it with the command rustup default nightly.

Calling Rust from other languages

Rust code can be called from any language that can call C. The Rust library should have the `crate-type` value `"dylib"`. When `rustfn1` is the Rust function to be called, this must be declared as:

```
#[no_mangle]
pub extern "C" fn rustfn1() { }
```

`#[no_mangle]` serves to keep the function names plain and simple, so that they are easier to link to. The `"C"` exports the function to the outside world with the C calling convention.

Some examples to call Rust from C itself, Python, Haskell, and Node.js can be found in this article `https://zsiciarz.github.io/24daysofrust/book/vol1/day23.html`. How to call Rust from Perl and Julia is detailed at `http://paul.woolcock.us/posts/rust-perl-julia-ffi.html`.



For a nice collection of examples, see `http://jakegoulding.com/rust-ffi-omnibus/`.

Summary

In this chapter, we proceeded into unsafe territory, where raw pointers point the way. We covered how to use assembly code, how to call C functions from Rust, and how to call Rust functions from other languages.

This chapter concludes our essential tour of Rust. In the upcoming chapters, we will look at the standard library and the crates ecosystem, providing pointers (no pun intended) for your Rust journey.

11
Exploring the Standard Library

Throughout this book, we have explained and used the concepts of Rust's Standard Library--the `std` crate. In this chapter, we will be discovering important standard modules that were not yet been covered (such as working with paths and files), and get a better grasp of working with collections.

We will discuss the following topics:

- Exploring `std` and the `prelude` module
- Collections - using hashmaps and hashsets
- Working with files
- Using Rust without the Standard Library

Exploring std and the prelude module

In previous chapters, we used built-in modules such as `str`, `vec`, and `io` from the `std` crate. The `std` crate contains many modules and functions that are used in real projects. For that reason, you won't see `extern crate std`, because `std` is imported by default in every other crate.

The small `prelude` module in `std` declares mostly traits (such as `Copy`, `Send`, `Sync`, `Drop`, `Clone`, `Eq`, `Ord`, and so on) and common types (such as `Option` and `Result`). For the same reason, the contents of the `prelude` module are imported by default in every module, as well as a number of standard macros (such as `println!`). That is the reason why we don't need to specify `Result::` before the `Ok` and `Err` variants in `match` branches.

The website `https://doc.rust-lang.org/std/` shows lists of the primitive types, modules, and macros contained in the standard library. We already discussed the most important macros from the standard library; see `Chapter 5`, *Higher-Order Functions and Error-Handling*, and `Chapter 8`, *Organizing Code and Macros*, in the *Some other built-in macros* section.

Collections - using hashmaps and hashsets

Often, you need to collect a large number of data items of a given type in one data structure. Until now, we have only worked with the `Vec` datatype to do this, but the `std::collections` module contains other implementations of sequences (such as `LinkedList`), maps (such as `HashMap`), and sets (such as `HashSet`). In most cases, you will only need `Vec` and `HashMap`; let's examine how to work with the latter.

A `HashMap <K,V>` is used when you want to store pairs consisting of a key of type `K` and a value of type `V`, both of generic type. `K` can be a Boolean, an integer, a string, or any other type that implements the `Eq` and `Hash` traits. A `HashMap` enables you to very quickly look up the value attached to a certain key using a hashing algorithm, and also because the data is cached. However, the keys of a `HashMap` are not sorted; if you need that, use a `BTreeMap`. `HashMap` is allocated on the heap, so it is dynamically resizable, just like a `Vec`.

Let's store monster names together with the planets they originate from:

```
// code from Chapter 11/code/hashmaps.rs:
use std::collections::HashMap;

fn main() {
    let mut monsters = HashMap::new();

    monsters.insert("Oron", "Uranus");
    monsters.insert("Cyclops", "Venus");
    monsters.insert("Rahav", "Neptune");
    monsters.insert("Homo Sapiens", "Earth");

    match monsters.get(&"Rahav") {
        Some(&planet) =>
println!("Rahav originates from: {}", planet),
        _ => println!("Planet of Rahav unknown."),
    }

    monsters.remove(&("Homo Sapiens"));

    // `HashMap::iter()` returns
    for (monster, planet) in monsters.iter() {
```

```
println!("Monster {} originates from planet {}", monster, planet);
    }
}
```

This program prints out:

```
Rahav originates from: Neptune
Monster Rahav originates from planet Neptune
Monster Cyclops originates from planet Venus
Monster Oron originates from planet Uranus
```

A new `HashMap` with default capacity is created with `HashMap::new()`. `insert` is used to put data into it, one pair at a time. If the inserted value is new, it returns `None`, otherwise, it returns `Some(value)`. To retrieve a value (here the planet on which a monster comes from), use the `get` method. This takes a reference to a key value and returns `Option<&V>`, so you have to use `match` to find the value. `remove` takes a key and removes the pair from the collection. The `iter()` method from `HashMap` returns an iterator that yields `(&key, &value)` pairs in arbitrary order.

If your requirement is to store unique values, such as unique monster names, `HashSet` guarantees that; it is in fact `HashMap` without values:

```
// code from Chapter 11/code/hashsets.rs:
use std::collections::HashSet;

fn main() {
    let mut m1: HashSet<&str> = vec!["Cyclops", "Raven",
"Gilgamesh"].into_iter().collect();
    let m2: HashSet<&str> = vec!["Moron", "Keshiu",
"Raven"].into_iter().collect();

    m1.insert("Moron");
    if m1.insert("Raven") {
        println!("New value added")
    }
    else {
        println!("This value is already present")
    }
    println!("m1: {:?}", m1);

println!("Intersection: {:?}", m1.intersection(&m2).collect::<Vec<_>>());
    println!("Union: {:?}", m1.union(&m2).collect::<Vec<_>>());
 println!("Difference: {:?}", m1.difference(&m2).collect::<Vec<_>>());
    println!("Symmetric Difference: {:?}",
             m1.symmetric_difference(&m2).collect::<Vec<_>>());
}
```

This program prints out:

```
This value is already present
m1: {"Raven", "Cyclops", "Gilgamesh", "Moron"}
Intersection: ["Raven", "Moron"]
Union: ["Raven", "Cyclops", "Gilgamesh", "Moron", "Keshiu"]
Difference: ["Cyclops", "Gilgamesh"]
Symmetric Difference: ["Cyclops", "Gilgamesh", "Keshiu"]
```

We can construct a `HashSet` from a `Vec` using `into_iter().collect()`. To add new values one by one, use `insert()`, which returns `true` if the value is new, or `false` if it is not. If the type of the set's elements implements the `Debug` trait, we can print them out with the `{:?}` format string. Methods for the intersection, union, difference, and symmetrical difference operations are also present.

Working with files

Now we show you how to do common operations with files, such as reading and writing, and making folders, and at the same time do proper error handling. The modules that contain these functionalities are `std::path`, which provides for cross platform file path manipulation, and `std::fs`.

Paths

File paths from the underlying file system are represented by the `Path` struct, which is created from a string slice containing the full path with `Path::new`. Suppose `hello.txt` is an existing file in the current directory; let's write some code to explore it:

```
// code from Chapter 11/code/paths.rs:use std::path::Path;

fn main() {
    let path = Path::new("hello.txt");
 let display = path.display();
    // test whether path exists:
    if path.exists() {
        println!("{} exists", display);
    }
    else {
        panic!("This path or file does not exist!");
    }

    let file = path.file_name().unwrap();
```

```
let extension = path.extension().unwrap();
let parent_dir = path.parent().unwrap();
    println!("This is file {:?} with extension {:?} in folder {:?}", file,
extension, parent_dir);

    // Check if the path is a file:
    if path.is_file() { println!("{} is a file", display);  }
    // Check if the path is a directory:
    if path.is_dir() { println!("{} is a directory", display); }
}
```

This code performs the following:

- Prints: `This is file "hello.txt" with extension "txt" in folder ""`.
- The `Path::new("e.txt").unwrap_or_else()` method creates the file `e.txt` or panics with an error.
- Tests whether a path is valid. Use the `exists()` method.
- Uses the `create_dir()` method to make folders.
- The `file_name()`, `extension()` and `parent()` methods return an Option value, so use `unwrap()` to get the value. The `is_file()` and `is_dir()` methods test respectively whether a path refers to a filename or to a directory.
- The `join()` method can be used to build paths, and it automatically uses the operating system-specific separator:

```
let new_path = path.join("abc").join("def");

// Convert the path into a string slice
match new_path.to_str() {
        None => panic!("new path is not a valid UTF-8 sequence"),
        Some(s) => println!("new path is {}", s),
      }
```

- This prints out the following on a Windows system:

```
 new path is hello.txt\abc\def
```

Reading a file

The `File` struct from `std::fs` represents a file that has been opened (it wraps a file descriptor), and gives read and/or write access to the underlying file.

Since many things can go wrong when doing file I/O, explicit and proactive handling of all possible errors is certainly needed. This can be done with pattern matching, because all the `File` methods return the `std::io::Result<T>` type, which is an alias for `Result<T, io::Error>`.

To open a file in read-only mode, use the static `File::open` method with a reference to its path, and match the file handler or a possible error like this:

```
// code from Chapter 11/code/read_file.rs:
use std::path::Path;
use std::fs::File;
use std::io::prelude::*;
use std::error::Error;

fn main() {
    let path = Path::new("hello.txt");
    let display = path.display();

    let mut file = match File::open(&path) {
        Ok(file) => file,
        Err(why) => panic!("couldn't open {}: {}", display,
Error::description(&why))
    };
}
```

In the case of a failing open method (because the file does not exist or the program has no access) this prints out:

```
thread '<main>' panicked at 'couldn't open hello999.txt: os error',
F:\Rust\Rust book\Chapter 11 - Working with files\code\read_file.rs:11
```

Explicitly closing files is not necessary in Rust; the `file` handle goes out of scope at the end of main or the surrounding block, causing the file to automatically close.

We can read in the file in a `content` string by using the `read_to_string()` method, which again returns a `Result` value, necessitating a pattern match. In order to be able to use this, we need to do the import `use std::io::prelude::*`:

```
 let mut content = String::new();
    match file.read_to_string(&mut content) {
        Err(why) => panic!("couldn't read {}: {}",
display, Error::description(&why)),
        Ok(_) => print!("{} contains:\n{}", display, content),
    }
```

This prints out:

```
hello.txt contains:
"Hello Rust World!"
```

`std::io::prelude` imports common I/O traits in our code, such as `Read`, `Write`, `BufRead`, and `Seek`.

Error-handling with try!

In order to program in a more structured way, we'll use the `try!` macro, which will propagate the error upwards, causing an early return from the function, or unpack the success value. When returning the result, it needs to be wrapped in either `Ok` to indicate success or `Err` to indicate failure. This is shown here:

```
// code from Chapter 11/code/read_file_try.rs:
use std::path::Path;
use std::fs::File;
use std::io::prelude::*;
use std::error::Error;
use std::io;

fn main() {
    let path = Path::new("hello.txt");
    let display = path.display();

    let content = match read_file(path) {
         Err(why) => panic!("error reading {}: {}", display,
Error::description(&why)),
         Ok(content) => content
    };

    println!("{}", content);
}
```

```
fn read_file(path: &Path) -> Result<String, io::Error> {
let mut file = try!(File::open(path));
let mut buf = String::new();
try!(file.read_to_string(&mut buf));
Ok(buf)
}
```

This prints out:

```
"Hello Rust World!"
```

Buffered reading

Reading in a file in memory with `read_to_string` might not be that clever with large files, because that would use a large chunk of memory. In that case, it is much better to use the buffered reading provided by `BufReader` and `BufRead` from `std::io`; this way lines are read in and processed one by one.

In the following example, a file with numerical information is processed. Each line contains two fields, an integer, and a float, separated by a space:

```
// code from Chapter 11/code/reading_text_file.rs:
use std::io::{BufRead, BufReader};
use std::fs::File;

fn main() {
    let file =
BufReader::new(File::open("numbers.txt").unwrap());
    let pairs: Vec<_> = file.lines().map(|line| {
        let line = line.unwrap();
        let line = line.trim();
        let mut words = line.split(" ");
let left = words.next().expect("Unexpected empty line!");
let right = words.next().expect("Expected number!");

(
left.parse::<u64>().ok().expect("Expected integer in first column!"),
right.parse::<f64>().ok().expect("Expected float in second column!")
)
}).collect();
     println!("{:?}", pairs);
}
```

This prints out:

```
[(120, 345.56), (125, 341.56)]
```

The information is collected in `pairs`, which is a `Vec<(u64, f64)>`, and which can then be processed as you want.

For a more complex line structure, you would want to work with struct values describing the line content instead of pairs, like this:

```
struct LineData {
    string1: String,
    int1 : i32,
    string2: String,
    // Some other fields
}
```

In production code, the `unwrap()` and `expect()` functions should be replaced by more robust code using pattern matching and/or `try!`.

Writing a file

To open a file in write-only mode, use the `File::create` static method with a reference to its path. This method also returns an `io::Result`, so we need to pattern match this value:

```
// code from Chapter 11/code/write_file.rs:
static CONTENT: &'static str =
"Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod
tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim
veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea
commodo consequat. Duis aute irure dolor in reprehenderit in voluptate
velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat
cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id
est laborum.
";

use std::path::Path;
use std::fs::File;
use std::io::prelude::*;
use std::error::Error;

fn main() {
    let path = Path::new("lorem_ipsum.txt");
    let display = path.display();

    let mut file = match File::create(&path) {
```

```
        Err(why) => panic!("couldn't create {}: {}",
                           display,
                           Error::description(&why)),
        Ok(file) => file,
    };
}
```

This command creates a new file or overwrites an existing one with that name, but it cannot create folders; if the path contains a subfolder, this must already exist. If you want to write a string (here a `static` string `CONTENT`) to that file, first convert the string with `as_bytes()` and then use the `write_all()` method, again matching the `Result` value:

```
match file.write_all(CONTENT.as_bytes()) {
        Err(why) => {
            panic!("couldn't write to {}: {}",
                            display,
                            Error::description(&why))
        },
        Ok(_) => println!("successfully wrote to {}", display),
}
```

This prints out:

```
successfully wrote to lorem_ipsum.txt
```

We need to write the string out as bytes; when writing a simple literal string like this to a file `file.write_all("line one\n");` we get `error: mismatched types: expected` `&[u8]`, found `&'static str`(expected slice, found str) [E0308]. To correct this, we must write (notice the b before the string for converting to bytes) this:

```
file.write_all(b"line one\n");
```

In fact, the `write_all` method continuously calls the `write` method until the buffer is written out completely. So we could just as well use `write`, and instead of a match we could also test on the successful outcome by checking `is_err`, like this:

```
if file.write(CONTENT.as_bytes()).is_err() {
println!("Failed to save response.");
return;
}
```

Instead of panicking in the case of an error, thus ending the program, we could also just display the error and return to the calling function, as in this variant:

```
let mut file = match File::create(&path) {
    Err(why) => { println!("couldn't create {}: {}",
                        display,
                        Error::description(&why));
                  return
                },
    Ok(file) => file,
};
```

Error-handling with try!

In the same way as we did for reading a file, we could also use the `try!` macro when writing to a file, like here:

```
// code from Chapter 11/code/write_file_try.rs:
use std::path::Path;
use std::fs::File;
use std::io::prelude::*;
use std::error::Error;
use std::io;

struct Info {
    name: String,
    age: i32,
    rating: i32
}

impl Info {
    fn as_bytes(&self) -> &[u8] {
        self.name.as_bytes()
    }

    fn format(&self) -> String {
        format!("{};{};{}\n", self.name, self.age, self.rating)
    }
}

fn main() {
    let path = Path::new("info.txt");
    let display = path.display();

    let file = match write_file(&path) {
        Err(why) => panic!("couldn't write info to file {}: {}",
```

```
                            display,
                            Error::description(&why)),
        Ok(file) => file,
    };
}

fn write_file(path: &Path) -> Result<File, io::Error> {    let mut file =
try!(File::create(path));
    let info1 = Info { name:"Barak".to_string(), age: 56, rating: 8 };
    let info2 = Info { name:"Vladimir".to_string(), age: 55, rating: 6 };
    try!(file.write(info1.as_bytes()));
    try!(file.write(b"\r\n"));
    try!(file.write(info2.as_bytes()));
    Ok(file)
}
```

To open a file in other modes, use the `OpenOptions` struct from `std::fs`.

Filesystem operations

Here is an example that shows some info for all files in the current directory:

```
// code from Chapter 11/code/read_files_in_dir.rs:
use std::env;
use std::fs;
use std::error::Error;

fn main() {
    show_dir().unwrap();
}

fn show_dir() -> Result<(), Box<Error>> {
    let here = try!(env::current_dir());
    println!("Contents in: {}", here.display());
    for entry in try!(fs::read_dir(&here)) {
        let path = try!(entry).path();
        let md = try!(fs::metadata(&path));        println!("  {} ({}
bytes)", path.display(), md.len());
    }
    Ok(())
}
```

This prints out:

```
Contents in: F:\Rust\Rust book\The Rust Programming Language\Chapter 11 -
Working with files\code F:\Rust\Rust book\The Rust Programming
Language\Chapter 11 - Working with files\code\read_file.rs (710 bytes)
F:\Rust\Rust book\The Rust Programming Language\Chapter 11 - Working with
files\code\read_files_in_dir.exe (2382143 bytes)
```

Here are some remarks to clarify the code:

- This code uses the `current_dir` method from `std::env`, reads that folder with `read_dir`, and iterates over all files it contains with `for entry in`
- To manipulate the filesystem, see `filesystem.rs` in the code download for examples, which demonstrates the use of the methods `create_dir`, `create_dir_all`, `remove_file`, `remove_dir`, and `read_dir`
- The `kind()` method is used here in the error case, to display the specific I/O error category (from the `std::io::ErrorKind` enum)
- The `fs::read_dir` method is used to read the contents of a directory, returning an `io::Result<Vec<Path>> paths`, which can be looped through with this code:

```
for path in paths {
    println!("> {:?}", path.unwrap().path());
}
```

Using Rust without the Standard Library

Rust is foremost a systems programming language and because the compiler can decide when a variable's lifetime ends, no garbage collection is needed for freeing memory. So when a Rust program executes, it runs in a very lightweight runtime, providing a heap, backtraces, stack guards, unwinding of the call stack when a panic occurs, and dynamic dispatching of methods on trait objects. Also, a small amount of initialization code is run before an executable project's `main` function starts up.

As we have seen, the standard library gives a lot of functionality. It offers support for various features of its host system: threads, networking, heap allocation, and more. It also links to its C equivalent, which also does some runtime initialization.

But Rust can also run on much more constrained systems that do not need (or do not have) this functionality. You can leave out the standard library from the compilation altogether by using the `#![no_std]` attribute at the start of the crate's code. In that case, Rust's runtime is roughly equivalent to that of C, and the size of the native code is greatly reduced.

When working without the standard library, the `core` crate and its `prelude` module are automatically made available to your code. The `core` library provides the minimal foundation needed for all Rust code to work. It comprises the basic primitive types, traits, and macros. For an overview, see; `https://doc.rust-lang.org/core/`.

At present, this only works for a library crate on the Rust stable version. Compiling an executable application without the standard library on Rust stable is much more involved; we refer you to `https://doc.rust-lang.org/unstable-book/language-features/lang-items.html`; `#using-libc` for a thorough and up-to-date discussion.

Summary

In this chapter, we showed you how to work with hashmaps and hashsets, two other important collection data structures. Then, we learned how to read and write files and explore the filesystem in Rust code. Finally, we looked at the Rust runtime, and how to make a project that runs without the standard library. We will conclude our overview of Rust's essentials in the next chapter by taking a closer look at Rust's ecosystem of crates.

12
The Ecosystem of Crates

Rust is a very rich language. This book has not discussed each and every concept of Rust, or covered it in great detail. Here we talk about things that have been left out and where the reader can find more information or details about these topics.

Furthermore, when starting off on a real project, you would certainly want to start by seeing what Rust code and crates already exist in that particular field. Gathering the most appropriate open source libraries from `https://crates.io/` to form the basis for your own application will certainly save you a lot of time!

In this chapter, we discuss the following topics:

- The ecosystem of crates
- Working with dates and times
- File formats and databases
- Web development
- Graphics and games
- OS and embedded system development
- Other resources for learning Rust

The ecosystem of crates

Compared to other languages, you will see that there is a tendency in Rust to move functionalities out of the core language into their own crates. It serves to keep the standard library small, thereby reducing the size of compiled apps. This is done, for example, for working with dates and times (see next section), or for providing special concurrency primitives.

An ever-growing ecosystem of crates for Rust is at your disposal at `https://crates.io/`, with nearing 11,000 crates in stock at the time of writing (Sep 2017). This repository site is also a showcase, because it is written in Rust! You can search for crates on specific keywords or categories, or browse them alphabetically or on the number of downloads. Find their documentation at `http://docs.rs`.

At *Awesome Rust* (`https://github.com/rust-unofficial/awesome-rust`), you can find a curated list of Rust projects; this site only contains useful and stable projects, and indicates whether they compile against the latest Rust version.

The *Rust Cookbook* (`https://rust-lang-nursery.github.io/rust-cookbook/`) provides a growing set of code recipes using common crates.

In general, it is advisable that you search for crates that are already available whenever you embark on a project that requires specific functionality. There is a good chance that a crate that conforms to your needs already exists, or perhaps you can find some usable starting code upon which to build exactly what you need. Take into consideration the number of downloads and when the crate was last updated; a popular and actively updated crate is most likely of higher quality.

Working with dates and times

The module `std::time` lets you work with the system time, and provides you with only the basic concepts of `Instant` and `Duration` (which we used in working with threads, see `Chapter 9`, *Concurrency - Coding for Multicore Execution*).

For any functionality beyond that, you need a crate. There are more than a dozen of them, with some specialized towards friendly output or games. The `chrono` crate is the most popular, however, and provides you with the most functionality, for example dates and times, timezones, and a wealth of formatting possibilities. Consult the docs for a complete overview `https://docs.rs/chrono/0.4.0/chrono/`.

Let's use a simple example to write the current local time to a file, applying what we learned in the previous chapter.

Start your project with `cargo new file_time --bin`.

In order to use the `chrono` crate, edit the dependencies section of your `Cargo.tml` to add this:

```
[dependencies]
chrono = "0.4"
```

Add the following to your crate root (here `main.rs`):

```
extern crate chrono;
use chrono::prelude::*;
```

Replace the code in `main.rs` with this:

```
let local: DateTime<Local> = Local::now();
println!("{}", local);
```

Then test this by giving the command `cargo run` in the `file_time` folder, which will download `chrono`, install and compile it, and then compile and run the `file_time` app. This provides the local time, for example:

```
2017-08-17 10:41:40.539165700 +02:00
```

We now format this to a somewhat prettier output such as `Thu, Aug 17 2017 10:41:40 AM`, and write it to a `log.txt` file with the `log_info` function. Here is the complete code of `main.rs`:

```
extern crate chrono;

use chrono::prelude::*;
use std::io::prelude::*;
use std::fs::File;
use std::io;

fn main() {
    let local: DateTime<Local> = Local::now();
    let formatted = local.format("%a, %b %d %Y %I:%M:%S %p\n").to_string();
    match log_info("log.txt", &formatted) {
        Ok(_) => println!("Time is written to file!"),
        Err(_) => println!("Error: could not create file.")
    }
}

fn log_info(filename: &str, string: &str) -> io::Result<()> {
    let mut f = try!(File::create(filename));
    try!(f.write_all(string.as_bytes()));
    Ok(())
}
```

The text `Time` is written to `file!`, which is printed on the terminal.

File formats and databases

The standard library offers the `std::fs` module for filesystem manipulation, which we explored in `Chapter 11`, *Exploring the Standard Library*.

- If you have to work with **Comma Separated Values** (**csv**) files, use one of the available crates such as `csv`, `quick_csv`, `simple_csv`, or `xsv`. The article on 24 days, `http://zsiciarz.github.io/24daysofrust/book/vol1/day3.html`, can get you started.
- For working with JSON files, use a crate such as `json, simple_json, serde_json`, or `json_macros`; start with reading `http://zsiciarz.github.io/24daysofrust/book/vol1/day6.html`.
- For XML format, there are also plenty of possibilities, such as the `xml-rs` or `quick-xml` crates.

For databases, there are many crates available for working with:

- **sqlite3** (crates `rusqlite` or `sqlite`)
- **PostgreSQL** (crates `postgres` and `tokio-postgres`); get started by using `http://zsiciarz.github.io/24daysofrust/book/vol1/day11.html`
- **Mysql** (crate `mysql` or `mysql_async`):

  Here is some code (`mysql.rs`) that connects to a local Mysql database, creates a `payment` table, and does some insert operations, and then finally reads back from that table:

```
#[macro_use]
extern crate mysql;

#[derive(Debug, PartialEq, Eq)]
struct Payment {
    customer_id: i32,
    amount: i32,
    account_name: Option<String>,
}

fn main() {
    let pool =
mysql::Pool::new("mysql://root:password@localhost:3307").unwrap();

    // Let's create a payment table.
    // It is temporary so we do not need `tmp` database to exist.
    // Unwap just to make sure no error happened.
```

```
    pool.prep_exec(r"CREATE TEMPORARY TABLE tmp.payment (
                          customer_id int not null,
                          amount int not null,
                          account_name text
                      )", ()).unwrap();

    let payments = vec![
 Payment { customer_id: 1, amount: 2, account_name: None },
 Payment { customer_id: 3, amount: 4, account_name: Some("foo".into()) },
 Payment { customer_id: 5, amount: 6, account_name: None },
 Payment { customer_id: 7, amount: 8, account_name: None },
 Payment { customer_id: 9, amount: 10, account_name: Some("bar".into()) },
    ];

    // Let's insert payments to the database
    // We will use into_iter() because we do not need to map Stmt to
anything else.
    // Also we assume that no error happened in `prepare`.
    for mut stmt in pool.prepare(r"INSERT INTO tmp.payment
     (customer_id, amount, account_name)
                           VALUES
     (:customer_id, :amount, :account_name)").into_iter() {
        for p in payments.iter() {
            // `execute` takes ownership of `params` so we pass account
name by reference.
            // Unwrap each result just to make sure no errors happened.
            stmt.execute(params!{
                "customer_id" => p.customer_id,
                "amount" => p.amount,
                "account_name" => &p.account_name,
            }).unwrap();
        }
    }

    // Let's select payments from database
    let selected_payments: Vec<Payment> =
    pool.prep_exec("SELECT customer_id, amount, account_name from
tmp.payment", ())
    .map(|result| { // In this closure we will map `QueryResult` to
`Vec<Payment>`
        // `QueryResult` is iterator over `MyResult<row, err>` so first
call to `map`
        // will map each `MyResult` to contained `row` (no proper error
handling)
        // and second call to `map` will map each `row` to `Payment`
        result.map(|x| x.unwrap()).map(|row| {
            let (customer_id, amount, account_name) = mysql::from_row(row);
            Payment {
```

```
                customer_id: customer_id,
                amount: amount,
                account_name: account_name,
            }
        }).collect() // Collect payments so now `QueryResult` is mapped to
`Vec<Payment>`
    }).unwrap(); // Unwrap `Vec<Payment>`
}
```

- **Oracle** (crate `oci_rs`)
- For **MongoDB** there is the `mongodb` crate, among many others, developed by the MongoDB lab:

  To get an idea how this works, here is some code (`mongodb.rs`) that connects to a local MongoDB and does some `insert`, `update`, `delete`, and `find` operations on a collection of `movies`:

```
let client = Client::connect("localhost", 27017)
    .expect("Failed to initialize client.");

let coll = client.db("media").collection("movies");
coll.insert_one(doc!{ "title" => "Back to the Future" }, None).unwrap();
coll.update_one(doc!{}, doc!{ "director" => "Robert Zemeckis" },
None).unwrap();
coll.delete_many(doc!{}, None).unwrap();

let mut cursor = coll.find(None, None).unwrap();
for result in cursor {
    if let Ok(item) = result {
        if let Some(&Bson::String(ref title)) = item.get("title") {
            println!("title: {}", title);
        }
    }
}
```

- For **Redis,** there are the `redis` or `simple_redis` crates; see `http://zsiciarz.github.io/24daysofrust/book/vol1/day18.html` for a quick start
- If you are interested in **Object Relational Mapper** (**ORM**) frameworks, the `diesel` crate is the most popular (see `http://zsiciarz.github.io/24daysofrust/book/vol2/day17.html`); it can be used for Sqlite, Mysql and Postgres; look at the `deuterium_orm` crate for a simpler framework

- If there is as yet no dedicated driver, you can probably use an ODBC connection through the `odbc-sys` or `odbc-safe` crates
- To use cypher queries for a neo4j database, use the `rusted_cypher` crate
- Furthermore, there are crates for leveldb, Cassandra (from Apache), RocksDB (from Facebook), Firebase, CouchDB, and InfluxDB

Web development

A general overview of the status of this domain can be found at `http://arewewebyet.com/`. At the time of writing, a number of web frameworks that provide vital support for basic needs are available. To get an initial idea of coding in practice, we have supplied a `"hello world"` example snippet for each of them:

- `iron` is the oldest framework. It was built for high concurrency needs, so it scales very well. The codebase has a high degree of modularity. Here is how you say `"hello-world"` in `iron`:

```
extern crate iron;

use iron::prelude::*;
use iron::status;

fn main() {
    Iron::new(|_: &mut Request| {
        Ok(Response::with((status::Ok, "Hello World!")))
    }).http("localhost:3000").unwrap();
}
```

- Another useful web framework inspired by `express.js` is `nickel` (`https://github.com/nickel-org/nickel.rs`). Say `"hello world"` in `nickel` like this:

```
#[macro_use] extern crate nickel;

use nickel::{Nickel, HttpRouter};

fn main() {
    let mut server = Nickel::new();
    server.get("**", middleware!("Hello World"));
    server.listen("127.0.0.1:6767");
}
```

- `conduit` provides a common HTTP server interface.

There is not yet the same level of functionality as the popular dynamically-typed web application frameworks (such as Rails, Phoenix, and Django) provide, but a few Rust frameworks, still under heavy development, are already emerging:

- `Rocket`, which has very good starting info and documentation on its website, `https://rocket.rs`. Say `"hello world"` in `Rocket` like this:

```
#![feature(plugin)]
#![plugin(rocket_codegen)]

extern crate rocket;

#[get("/")]
fn index() -> &'static str {
    "Hello, world!"
}

fn main() {
    rocket::ignite().mount("/", routes![index]).launch();
}
```

- `Gotham` (for more info, see `https://github.com/gotham-rs/`). Say `"hello world"` in `Gotham` like this:

```
extern crate futures;
extern crate hyper;
extern crate gotham;
extern crate mime;

use hyper::server::Http;
use hyper::{Request, Response, StatusCode};

use gotham::http::response::create_response;
use gotham::state::State;
use gotham::handler::NewHandlerService;

pub fn say_hello(state: State, _req: Request) -> (State, Response) {
    let res = create_response(
        &state,
        StatusCode::Ok,
        Some((
            String::from("Hello World!").into_bytes(),
            mime::TEXT_PLAIN,
        )),
    );
```

```
        (state, res)
    }

    pub fn main() {
        let addr = "127.0.0.1:7878".parse().unwrap();

        let server = Http::new()
            .bind(&addr, NewHandlerService::new(|| Ok(say_hello)))
            .unwrap();

        println!(
            "Listening on http://{}",
            server.local_addr().unwrap()
        );

        server.run().unwrap();
    }
```

If you only need a light micro-web framework, `rustful` could be your choice.

The most advanced and stable crate for developing HTTP applications at this moment is **`hyper`**. It is fast and contains both an HTTP client and server to build complex web applications. It is used by the `iron` web framework. To get started with it, read this introductory article `http://zsiciarz.github.io/24daysofrust/book/vol1/day5.html`.

For a lower-level library, you could start with `tiny_http`. `reqwest`, `curl`, and `tokio-curl` which are popular HTTP client libraries. If you need a **Representational State Transfer** (**REST**) framework, go for `rustless`.

And of course, don't ignore the new `servo` browser that is emerging!

Graphics and games

Its high performance and low-level capabilities make Rust an ideal choice in this field. Searching for graphics reveals bindings for OpenGL (`gl, glfw-sys`), Core Graphics, gfx, gdk, `gtk` (`http://gtk-rs.org`), or the minimal Gtk+ library `mg`, and others. `conrod` is a 2D GUI library and `relm` is an asynchronous, GTK+-based GUI library, inspired by Elm. `gtk` is the most advanced library, but at the time of writing a complete cross-platform GUI toolkit is not yet available.

Have a look at the `http://arewegameyet.com` website to see what the current status is on the Rust game front. There is a modular game engine called `piston` (`https://github.com/PistonDevelopers/piston`), chipmunk 2D, and bindings for SDL2 and Allegro5. `kiss3d` (see `http://kiss3d.org`) is a crate for a simple 3D game engine. A number of physics (such as `ncollide`) and math (such as `nalgebra` and `cgmath`) crates exist that can be of use here.

Here are the steps for a simple app that uses `piston` to draw a blue circle:

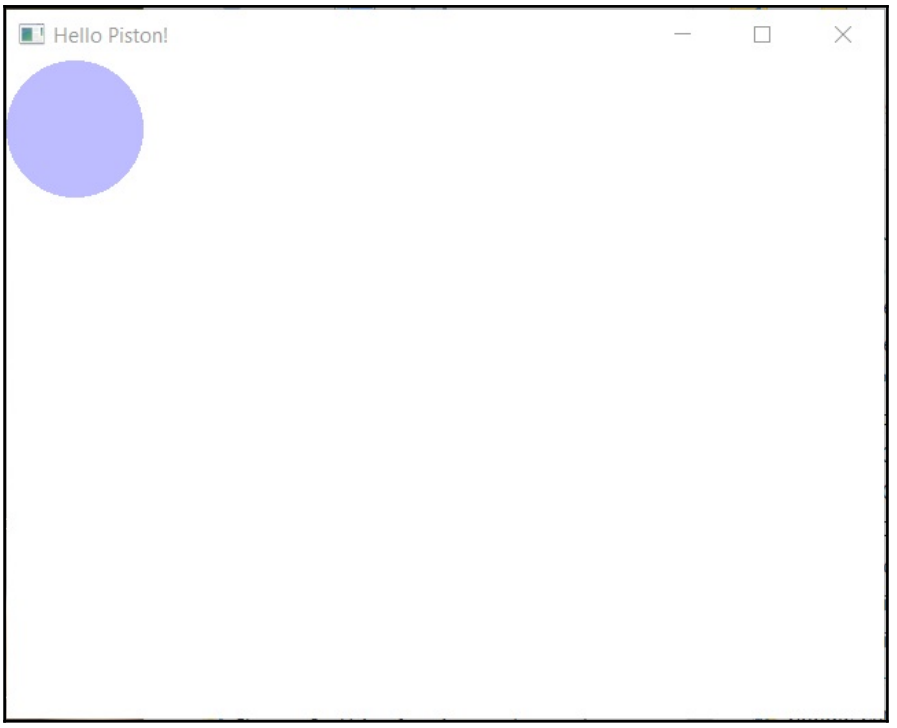
1. Create a new project with `cargo new piston101 –bin`.
2. Add `piston_window = "0.61.0"` to the `[dependencies]` section of `Cargo.toml`.
3. Replace the code in `src\main.rs` with this:

```
extern crate piston_window;

use piston_window::*;

fn main() {
         let mut window: PistonWindow =
        WindowSettings::new("Hello Piston!", [640, 480])
        .exit_on_esc(true).build().unwrap();
    while let Some(event) = window.next() {
    window.draw_2d(&event, |context, graphics| {
            clear([1.0; 4], graphics);
            ellipse([0.0, 0.0, 1.0, 0.5],
                    [0.0, 0.0, 100.0, 100.0],
                    context.transform,
                    graphics);
        });
    }
}
```

4. Do a `cargo run` in the `piston101` project folder. This will download tens of crates on which `piston_window` depends, compile them, compile the `piston101` app, and then the following window is shown:



OS and embedded system development

Several open source operating systems are written in Rust. Most of them are only proofs of concept, or suitable as demos in courses on operating systems. The only system that goes a step further is `redox` (see `https://www.redox-os.org`), which is actively being developed. It has a microkernel architecture and comes with a window manager, as well as basic applications such as an editor and file manager. You can find a comparison of several Rust OS projects, as well as a great number of useful links on this site `https://github.com/flosse/rust-os-comparison`.

`tock` (`https://www.tockos.org`) is an embedded operating system written in Rust and designed for running multiple concurrent, mutually distrustful applications on low-memory and low-power microcontrollers.

`cortex_m_rtfm` is a bare metal concurrency framework for Cortex-M microcontrollers geared towards robotics and control systems. `zinc` (`http://zinc.rs/`) is an example of a project that uses Rust to write a code stack for processors (currently ARM).

If you need async I/O in your project, look at `tokio` (https://tokio.rs) and the many related crates. At the core of `tokio` is the `futures` crate to model asynchronous computations. Read `https://lukesteensen.com/2016/12/getting-started-with-tokio/` to get you started.

Furthermore, crates exist for a lot of other categories, such as functional and embedded programming (see `http://spin.atomicobject.com/2015/02/20/rust-language-c-embedded/`), data structures, image processing (the `image` crate), audio, compression, encoding and encryption (such as `rust-crypto` and `crypto`), regular expressions, parsing, hashing, tooling, testing, template engines, and so on.

Have a look at crates.io or the *Awesome Rust* compilation (see `https://github.com/rust-unofficial/awesome-rust`) to get an idea of what is available.

Other resources for learning Rust

This book covered nearly all the topics of the so-called *Rust Book* (`http://doc.rust-lang.org/book/first-edition`), and sometimes went beyond. Nevertheless, the *Book* on the Rust website can still be a good resource for finding the latest info, together with a fine collection of Rust code examples at `http://rustbyexample.com`.



For the most complete in-depth information, consult the reference at `https://doc.rust-lang.org/stable/reference.html`.

Asking questions or following and commenting on the discussions on *Reddit* (`https://www.reddit.com/r/rust`) and *Stack Overflow* (`https://stackoverflow.com/questions/tagged/rust`) can also help you out. Last but not least, when you have an urgent Rust question, chat with the friendly experts on the IRC channel `https://client01.chat.mibbit.com/?server=irc.mozilla.org&channel=%23rust`.

A resource for coding guidelines on Rust can be found at `https://github.com/rust-lang-nursery/fmt-rfcs`. Most of these style rules are already implemented in the **rustfmt** tool, so it is better to develop a workflow to automatically use that tool when saving code in your preferred editor.

24 days of Rust is a highly recommended article series by Zbigniew Siciarz on a multitude of advanced Rust subjects and useful crates. Consult the index at `https://zsiciarz.github.io/24daysofrust/index.html`.

Here are some other nice collections of learning resources:

- `https://github.com/ctjhoa/rust-learning`: A very comprehensive overview of all kinds of learning materials (books, videos, podcasts, and so on)
- `https://hackr.io/tutorials/learn-rust`: A collection of tutorials

Summary

In this chapter, we showed you the richness of the Rust ecosystem, how to search for usable crates, and the most important crates you'll want to use in your development.

This brings us to the end of our Rust journey in this book. We hope you've enjoyed it as much as we enjoyed writing it. You now have a firm basis to start developing in Rust. We also hope that this whirlwind overview has shown you why Rust is a rising star in the software development world, and why you should take it up in your projects. Join the Rust community and start using your coding talents. Perhaps we'll meet again in the Rust (un)iverse.

Index

A

ACID2
  reference 15
adapters 97, 98, 99
aliasing 40
arrays
  about 69, 70
  and strings 73, 74, 75
assembly code
  inlining 210
associated functions
  on structs 110, 111
atomic reference counted pointer 194
atomic types 193
attributes 56

B

beta channel 12
borrowed pointer 143
borrowing 147
boxes 153, 154, 155, 156
branching
  on condition 47, 49
builder pattern
  using 115
built-in macro 176
built-in traits 125

C

C
  interfacing 207, 209
  library, using 209, 210
Cargo
  reference 20
cargo
  testing with 60
Cargo
  working with 19, 21
channels
  asynchronous 200
  creating 198
  data, receiving 197
  data, sending 197
  references, sending 199
  struct values, sending 199
  synchronous 200
  used, for communicating 196
closures 92, 93, 94, 95
collections 214
Comma Separated Values (csv) 230
comments 28
concurrency 185, 186
condition
  branching on 47, 49
conditional compilation 56, 57
console
  input, obtaining from 81, 82, 83
constructor pattern
  using 113
consumers 97, 98, 99
copy trait
  using 137, 138, 139, 140
crates
  about 162
  building 162, 163
  ecosystem 227, 228
  external crates, adding 174
  external crates, adding to project 172, 173
  external crates, importing 169, 170, 171
  macros, using 182
  public interface, exporting 171, 172
  random numbers, working with 174, 175
  reference 163, 172, 173, 228

custom conditions
  reference 57

D

dangling pointer 150
data races 7
data racing 192
databases 230
dates
  working with 228, 229
developer tools
  about 23
  Sublime Text 24
Don't Repeat Yourself (DRY) 177
Drop trait
  implementing 151
dynamic dispatch
  about 123
  reference 124

E

embedded system development 237, 238
encapsulation property 166
enums
  about 79
  algebraic data types 80
  methods 116
  Option 81
  Result 80
  union types 80
error crate
  reference 108
error-handling
  ? operator, using 107, 108
  about 101
  examples 104, 106
  panics 102
  testing, for failure 103, 104
  try! macro, using 223
  with try! macro 107, 108, 219
expressions 40, 41

F

feature gates 208
file formats 230
file paths 216
files
  buffered reading 220
  reading 218, 219
  working with 216
  writing 221
filesystem operations 224, 225
Foreign Function Interface (FFI) 204, 208
fragment specifier 178
functions
  about 51, 53, 99, 100, 101
  documenting 55
  example 52

G

generic data structures 99, 100, 101
global constants 29
Gotham
  reference 234
graphics
  and games 235, 236
gtk
  reference 235

H

hashmaps
  using 214
hashsets
  using 214
heap 42, 43, 44, 133
higher order functions 92, 93, 94, 95

I

immutable variables 35, 36
inheritance
  with traits 127, 128
input
  obtaining, from console 81, 82, 83
Integrated Development Environments (IDEs) 23
intellij-rust
  reference 25
iterators 95

J

just-in-time (JIT) 15

K

kiss3d
  reference 236

L

lifetime elision 135
lifetimes 133, 134, 135, 136
LLVM compiler framework
  reference 10
lock 193
looping 49, 50, 51

M

macros
  about 176
  built-in macros 183
  developing 177, 178
  examples 180, 181, 182
  function, creating 180
  reference 179
  repetition 179, 180
  using 176, 177
  using, from crates 182
Markdown
  reference 56
match keyword 146
matcher 177
matching patterns 84, 86, 87
methods
  on enums 116
  on structs 112, 113
  on tuples 116
modules
  about 162, 164
  defining 164, 166
  file hierarchy 167, 168
  importing 167, 168
  public interface, exporting 171, 172
  visibility 165, 167
MongoDB 232
moving closure 152, 153
Mozilla 8
mutable variables 35, 36
mutex 193

N

nested modules 164
nightly channel 12

O

Object Relational Mapper (ORM) 232
OOP
  in Rust 127
Operating System (OS) 186
operator overloading 125
ownership 147

P

Package Control package
  installation link 24
piston
  reference 236
pointers 132, 141, 159
prelude module
  exploring 213
primitive types 32, 33
Private Implementation Pattern (PIMPL) 167
program 17, 18
program arguments 89

R

raw pointers 206, 207
Read-Evaluate-Print-Loop (REPL) 25
Redis 232
RedMonk ranking of programming language
  reference 14
redox
  reference 237
ref keyword 146
reference counting 157, 158
references 132, 143, 144, 145
Representational State Transfer (REST) 235
Resource Acquisition Is Initialization (RAII)
  reference 151
Rocket

reference 234
Rust documentation
consulting 33
rust-netbeans
reference 25
Rust
about 7, 9, 12
advantages 8, 9, 13
calling 211
comparison, with other languages 11
installation link 16
installing 16
learning resources 238, 239
major release 12
minor release 12
patch release 12
reference 8, 14, 33
references 212
using 15
using, without Standard Library 225, 226
rustaceans 10
rustc 17
RustDT
reference 25
rustfmt tool 238

S

semantic versioning
reference 173
Servo
about 15
reference 15
shadowing 37
shared library 163
shared memory 192
shared mutable states
about 193, 194
Sync trait 195
shared ownership 157
slices 73
stable channel 12
stack 42, 43, 44, 133
stainless crate
reference 63
Standard Library 26
static dispatch 123
std module
exploring 213
string interpolation
printing with 30, 31
string slices 65
strings
about 65, 66
and arrays 73, 74, 75
struct keyword 146
structs
about 77, 79
associated functions 110, 111
methods 112, 113
Sublime Text
about 24
reference 24
using 25
Sync trait 195

T

testing
about 57, 58, 59
with cargo 60
tests module 61, 62, 63
thread-safe 192
threads
about 185
creating 186, 187, 188
generating 189, 190, 191
panicking 191
safety 192
stack size, setting 188
times
working with 228, 229
TIOBE index
reference 13
tock
reference 237
tokio
reference 238
TOML format
reference 21
trait constraints
using 120, 121, 122

trait objects 123
traits 11, 117, 118, 119, 120
try! macro
  using, in error handling 219
tuples
  about 75, 76
  methods 116
type checking 37, 38, 39
type conversions 37, 38, 39

U

unsafe block
  about 203, 204
  std mem, using 205

V

value types 32, 33
value
  borrowing 148, 149, 150
  moving 148
  printing, with string interpolation 30, 31
  variables, binding to 34, 35
variables
  binding, to values 34, 35
  scope 37
vectors 71, 72, 73
visitor pattern
  reference 129
  using 129
VisualRust
  reference 25

W

web development 233, 234

Z

zinc
  reference 17, 237