Code Like a Pro in Rust

Brenden Matthews

MEAP

MANNING

MEAP Edition
Manning Early Access Program
Code Like a Pro in Rust
Version 10

For more information on this and other Manning titles go to
manning.com

welcome

Thanks for purchasing the MEAP for *Code Like a Pro in Rust*. Rust is one of the most loved programming languages for good reason, and I'm confident you'll love it too once you've learned how to get past the prickly parts.

To get the most out of this book, you should be familiar with the Rust programming language, but you need not be an expert. It would be helpful, for your benefit, if you consider yourself an expert in at least one other programming language. If you *are* an expert in Rust, some of this book may be redundant, but there are plenty of fresh new ideas in this book for you, too.

This is *not* an introductory book for the Rust language; this book is for those who've learned the basics of the language and want to fast forward past the *bang-your-head-in-frustration* phase of learning a new programming language. If you're new to programming in general, you may find this book quite challenging, and while I won't discourage you from reading it, you may find it difficult to follow.

The Rust language provides a *plethora* of tools and features that can help you stay in a flow state, make programming a pleasure, and help you produce high quality, safe, and high-performance software. With Rust, however, you may find the new jargon and tooling sometimes leaves you with head-spinning confusion, and perhaps a bit of sea sickness. This book is like a life raft packed full of juicy knowledge, and it's here to help as you drift around the sea of confusing new terminology, tooling, and language concepts you'll encounter in the Rust ecosystem.

There's a lot of exciting stuff to discover on your way to becoming a Rust pro, and with this book you'll quickly learn the tools and conventions you'll need to know to ensure Rust sparks joy.

Your feedback is invaluable to me and to others who read this book. Your comments and questions in the liveBook discussion forum will help me find the blind spots and fill the book's knowledge gaps. We're going to make an awesome Rust book.

—Brenden Matthews

brief contents

1 Feelin' Rusty

This chapter covers

- A brief introduction to Rust
- Overview of the language and its purpose
- Comparing Rust to other programming languages
- How to get the most out of this book

This book will help new Rust developers get up to speed on the language, tooling, design patterns, and best practices as quickly as possible. By the end of this book, you should feel confident building production grade software systems with idiomatic–or *Rustaceous*–Rust. This book is not an exhaustive reference of the Rust language or its tooling; instead, this book focuses on just the the good stuff. For readers who *aren't* new to Rust, you'll likely still find the content valuable to enhance your skills.

Rust offers compelling features for those looking to build fast, safe programs. Some people find Rust's learning curve a bit steep, and this book can help overcome the challenging parts, clarify Rust's core concepts, and provide actionable advice.

The book is written for those already familiar with the Rust programming language. Additionally, it will be of much benefit to the reader to have experience with other system-level programming languages such as C, C++, or Java. You need not be an expert in Rust to get value out of this book, but I won't spend much time reviewing basic syntax, history, or programming concepts.

Many of the code samples in this book are partial listings, but the full working code samples can be found on GitHub at https://github.com/brndnmtthws/code-like-a-pro-in-rust-book. The code is made available under the MIT license, which permits usage, copying, and modifications without

restriction. I recommend you follow along the full code listings if you can, to get the most out of this book. The code samples are organized by chapter within the repository, however some examples may span multiple sections or chapters, and are thus named based on their subject matter.

1.1 What's Rust?

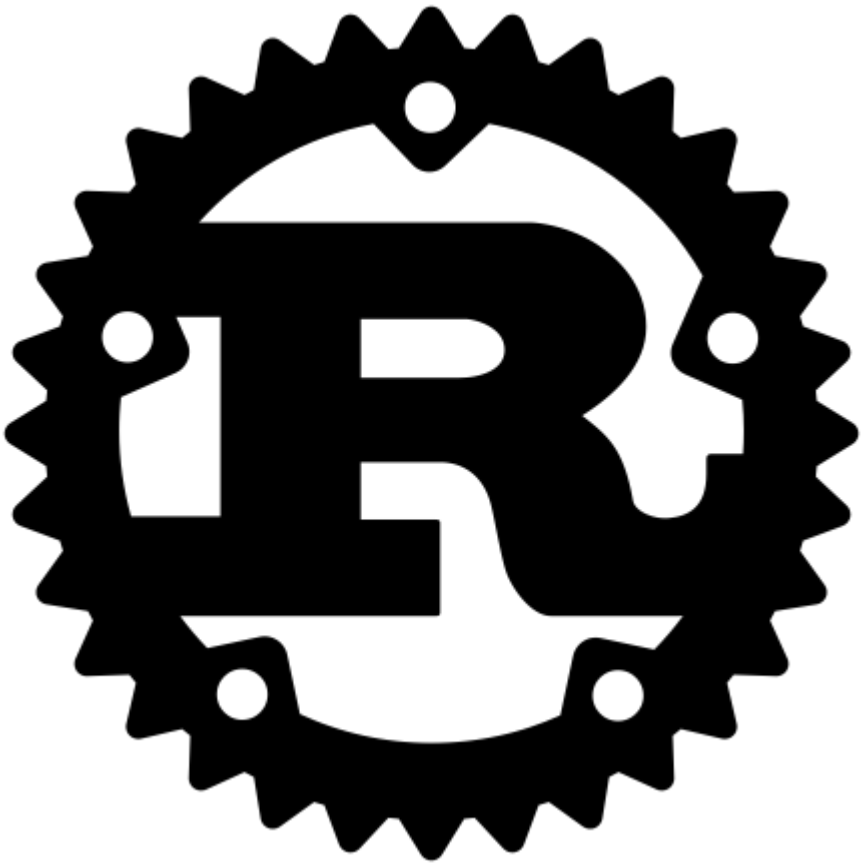

Figure 1.1 Rust language logo, by the Rust core team, CC BY 4.0

Rust is a modern programming language with a focus on performance and safety. It has all the features you may want or expect in a modern programming language like closures, generics, asynchronous I/O, powerful tooling, IDE integrations, linters and style checking tools, as well as a vibrant growing community of developers and contributors.

Rust is a powerful language and can be used for many different things, including web development. While it was written with the intention of being a systems level language, it also fits quite well in domains that are well outside system level programming, such as web programming with *Wasm* (short for WebAssembly, a web standard for executing bytecode). In figure 1.2 I've illustrated where Rust typically sits in the language stack, but this is by no means a definitive definition.

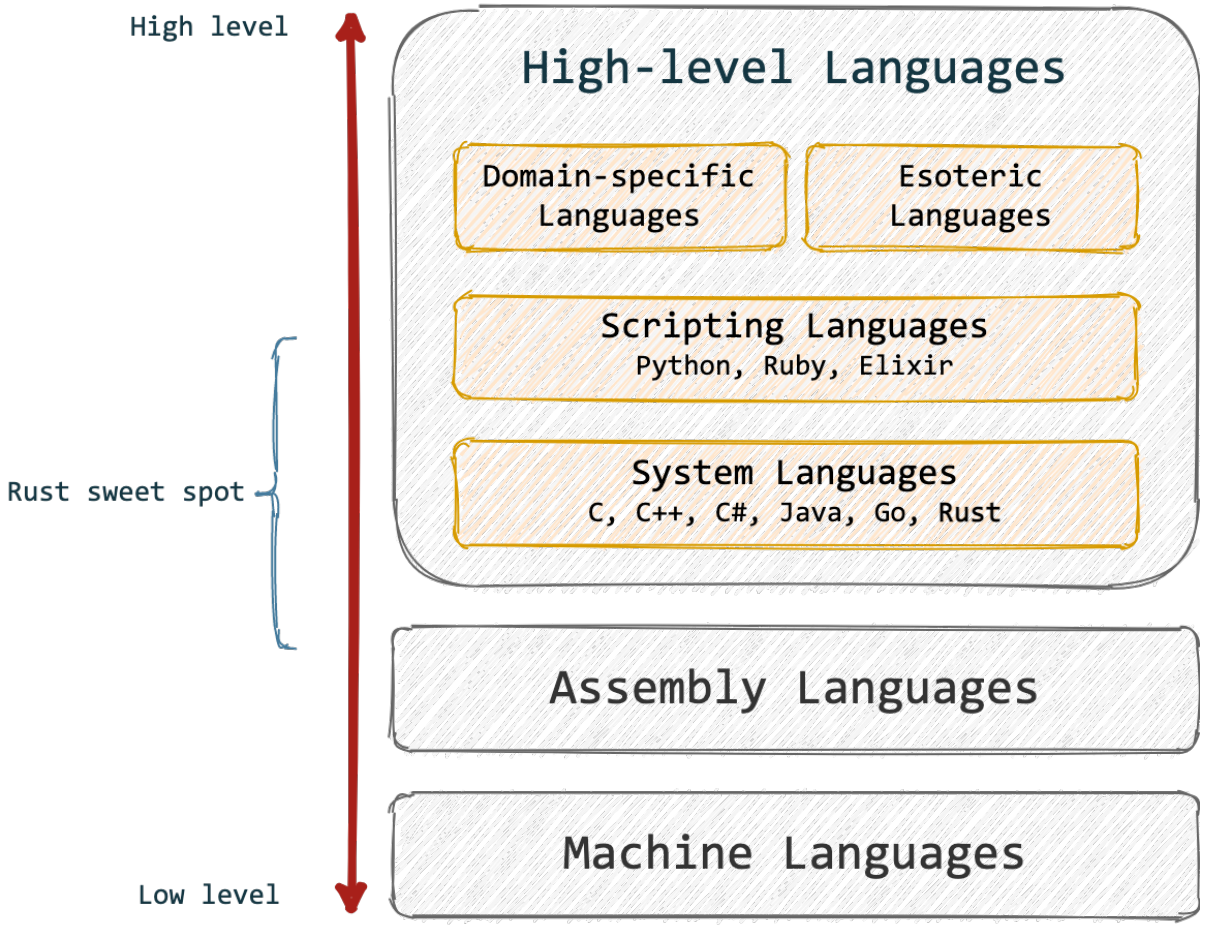

Figure 1.2 Where Rust fits in language classifications

Rust's creators envisioned its primary use as building system-level code and libraries which are safety and performance critical. Rust's safety guarantees don't come for free, the cost of those features comes in terms of added language and compilation-time complexity.

Rust *can* compete with higher level languages like Python or Ruby, however the main differentiator is the lack of a runtime interpreter, as Rust is compiled to platform-dependent binaries. Thus, one must distribute their Rust programs as binaries (or somehow provide a compiler). There are a few particular cases where Rust is likely a much better choice than a scripting language like Python or Ruby, such as embedded or resource constrained environments.

Rust can also be compiled for web browsers directly through the use of Wasm, which has grown significantly in popularity recently. Wasm is simply treated as yet another CPU target, much like x86-64 or aarch64, except the CPU in this case is a web browser.

Some highlights of the Rust language are:

- A core suite of tools for working with the language, including but not limited to:
  - `rustc`, the official Rust compiler
  - `cargo`, a package manager and build tool

    - https://crates.io, a package registry
- Many modern programming language features, including:
    - The borrow checker, which enforces Rust's memory management model
    - Static typing
    - Asynchronous I/O
    - Closures
    - Generics
    - Macros
    - Traits
- Numerous community tools for improving code quality and productivity:
    - `rust-clippy`, an advanced linter and style tool
    - `rustfmt`, an opinionated code formatter
    - `sccache`, a compiler cache for `rustc`
    - `rust-analyzer`, full-featured IDE integration for the Rust language

SIDEBAR **Most loved language**

Rust has won Stack Overflow's annual developer survey[1] in the category of "most loved programming language" every year since 2016, as of the time of writing.

In the 2021 survey, out of 82,914 responses Rust was loved by 86.98% of those using it. The second place language, Clojure, came in at 81.12% loved, and the third place language, TypeScript, came in at 72.73% loved.

1.2 What's unique about Rust?

Rust addresses common programming mistakes with a unique set of abstractions, some of which you may have never encountered before. In this section, I'll provide a quick tour of the features that make Rust different.

1.2.1 Rust is safe

Safety is one of Rust's spotlight features. Rust's safety features are its biggest differentiator from most other languages. Rust can provide strong safety guarantees thanks to a feature called the *borrow checker*.

In languages like C and C++, memory management is a somewhat manual process, and developers must be aware of the implementation details when considering memory management. Languages like Java, Go, and Python use automatic memory management, or garbage collection, which obfuscate the details of allocating and managing memory with the trade off of incurring some performance overhead.

Rust's borrow checker works by validating references at *compile time*, rather than reference counting or performing garbage collection at runtime. It's a unique feature that also introduces

challenges when writing software, especially if you've never encountered the borrow checker.

The borrow checker is part of Rust's compiler `rustc`, which verifies that for any given object or variable, there can be no more than one mutable reference at a time. It's possible to have multiple immutable references (i.e., read-only references) to objects or variables, but you may never have more than a single active mutable reference.

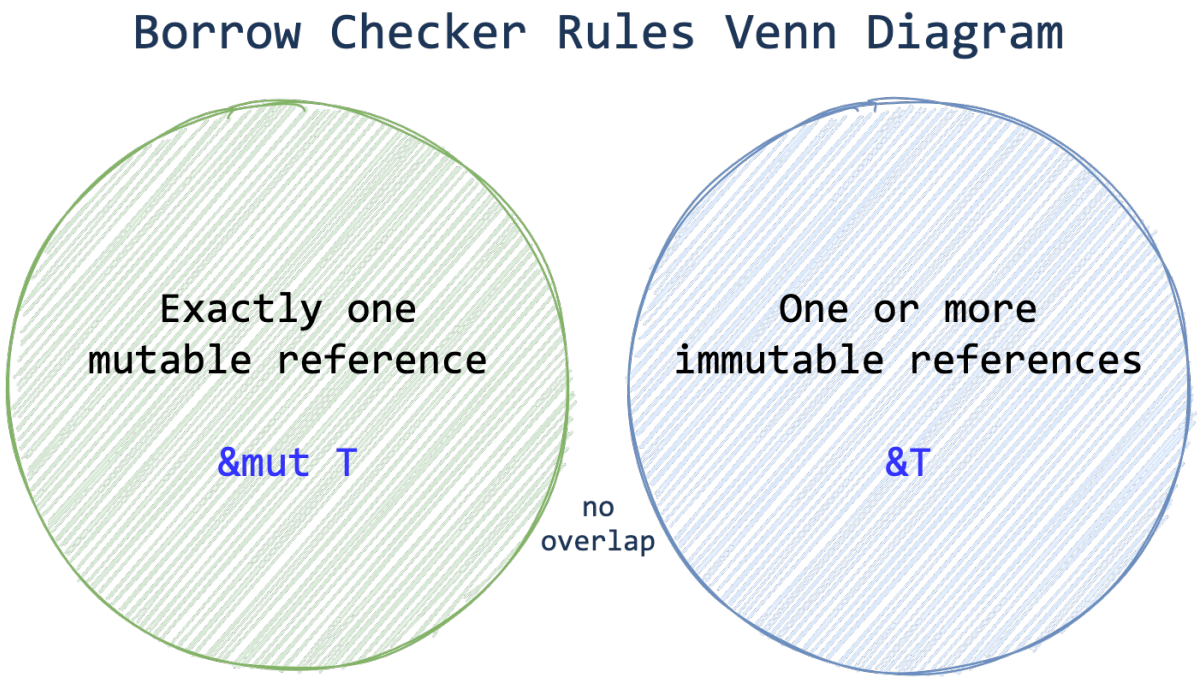

Figure 1.3 Rust borrow checker rules Venn diagram

Rust uses *RAII* (resource acquisition is initialization) to keep track of when variables and all their references are in and out of scope. Once they are out of scope, memory can be released. The borrow checker will not allow references to out of scope variables, and it only allows one mutable reference or multiple immutable references, but never both.

The borrow checker provides safety for parallel programming, too. Race conditions arise when sharing data in parallel, such as between separate threads. In most cases the root cause is the same: simultaneous shared mutable references. With Rust, it's not possible to have more than one mutable reference, thereby ensuring data synchronization issues are avoided, or at least not created unintentionally.

Rust's borrow checker is tricky to master at first, but soon you'll find it's the best feature of Rust. Similar to languages like Haskell, once you manage to make your code compile, that's often enough (when combined with adequate testing) to guarantee your code will work and never crash (testing is covered in chapters 6 & 7). There are exceptions to this, but by and large code written in Rust will not crash from common memory errors like reading past the end of a buffer or mishandling memory allocations and deallocations.

1.2.2 Rust is modern

The Rust language developers have paid special attention to supporting modern programming paradigms. Coming from other languages, you may notice Rust's out-with-the-old, in-with-the-new approach. Rust largely eschews paradigms like object-oriented programming in favour of traits, generics, and functional programming.

Notably, Rust emphasisizes the following paradigms and features:

- Functional programming: closures, anonymous functions, iterators
- Generics
- Traits, sometimes referred to as interfaces in other languages
- Lifetimes, for handling references
- Metaprogramming through its macro system
- Asynchronous programming, with async/await
- Package and dependency management with Cargo
- Zero cost abstractions

Traditional object-oriented features are notably absent from Rust. And while it's true that you can model patterns similar to classes and inheritence in Rust, the terminology is different, and Rust lends itself toward functional programming. For those coming from an object-oriented background such as C++, Java, or C#, it may take some time to get used to. Many programmers, once they adjust to the new patterns, find themselves with a certain delight and freedom in being liberated from the rigidness of object-oriented ideology.

1.2.3 Rust is pure open source

When considering languages and platforms to build upon, community governance is an important thing to consider when thinking about the long term maintenance of any project.

Some languages and platforms that are open source but mostly governed by large companies such as Go (Google), Swift (Apple), .NET (Microsoft) come with certain risks, such as having the platforms that govern the projects make decisions about which technologies to include or support which favour their products.

Rust is a community-driven project, lead primarily by the not-for-profit Mozilla Foundation. The Rust programming language itself is dual-licensed under the Apache license and the MIT license. Individual projects within the Rust ecosystem individually licensed, but most key components and libraries exist under open source licenses, such as MIT or Apache.

There is strong support amongst large technology companies for Rust. Amazon, Facebook, Google, Apple, Microsoft, and others have made plans to use or pledge support for Rust. By not being tied to any particular entity, Rust is a good long-term choice with minimal conflict of interest.

NOTE The Rust team maintains a list of production users on the official Rust language website at https://www.rust-lang.org/production.

1.2.4 Rust versus other popular languages

The table below, while not exhaustive, provides a summary of differences between Rust and other popular programming languages.

Table 1.1 Rust compared to other languages

Language	Paradigm	Typing	Memory model	Key features
Rust	multi: concurrent, functional, generic, imperative	static, strong	RAII, explicit	safety, performance, async
C	imperative	static, weak	explicit	efficiency, portability, low-level memory management, widely supported
C++	multi: imperative, object-oriented, generic, functional	static, mixed	RAII, explicit	efficiency, portability, low-level memory management, widely supported
C#	multi: object-oriented, imperative, event-driven, functional, reflective, concurrent	static, dynamic, strong	garbage collected	supported on Microsoft platforms, large ecosystem, advanced language features
JavaScript	multi: prototypes, functional, imperative	dynamic, duck, weak	garbage collected	widely supported, async
Java	multi: generic, object-oriented, imperative, reflective	static, strong	garbage collected	bytecode-based, production grade Java Virtual Machine, widely supported, large ecosystem
Python	multi: functional, imperative, object-oriented, reflective	dynamic, duck, strong	garbage collected	interpreted, highly portable, widely used
Ruby	multi: functional, imperative, object-oriented, reflective	dynamic, duck, strong	garbage collected	syntax, everything's an expression, simple concurrency model
TypeScript	multi: functional, generic, imperative, object-oriented	static, dynamic, duck, mixed	garbage collected	types, JavaScript compatibility, async

1.3 When should you use Rust?

Rust is a systems programming language, generally meant to be used for lower level system programming, in situations similar to where you'd use C or C++. Rust may not be well suited for use cases where you want to optimize for developer productivity, as writing Rust can often be trickier than writing code with popular languages like Go, Python, Ruby, or Elixir.

Rust is also a great candidate for web programming with the onset of WebAssembly. You can build applications and libraries with Rust, compile them for Wasm, and take advantage of the benefits of Rust's safety model with the portability of the web.

There is no specific use case for Rust. You should use it where it makes sense. I have created a list of example use cases where Rust is well suited. I have personally used Rust for many small

one-off projects, simply because it's a joy to write and once the code compiles you can generally count on it working. With proper use of the Rust compiler and tooling, your code is considerably less likely to have errors or behave in undefined ways–which is a desireable property for *any* project.

YOUR TURN **Using the right tool for the right job is important for any endeavour to succeed, but in order to know which tools are the right tools, you must first gain experience using a variety of different tools for different tasks.**

1.3.1 Rust use cases

- **Code Acceleration**
  Rust can accelerate functions from other languages like Python, Ruby, or Elixir.
- **Concurrent systems**
  Rust's safety guarantees apply to concurrent code. This makes Rust ideal for use in high performance concurrent systems.
- **Cryptography**
  Rust is ideally suited for implementing cryptography algorithms.
- **Embedded programming**
  Rust generates binaries that bundle all dependencies, excluding the system C library or any 3rd party C libraries. This lends itself to relatively straight forward binary distribution, particularly on embedded systems. Additionally, Rust's memory management model is great for systems that demand minimal memory overhead.
- **Hostile environments**
  In situations where safety is of utmost concern, Rust's guarantees are a perfect fit.
- **Performance critical**
  Rust is optimized for safety *and* performance. It's easy to write code that's extremely fast, without compromising on safety with Rust.
- **String processing**
  String processing is an incredibly tricky problem, and Rust is particularly well suited for the task because it's easy to write code that can't overflow.
- **Replacing legacy C or C++**
  Rust is an excellent choice for replacing legacy C or C++.
- **Safe web programming**
  Rust can target WebAssembly, allowing us to build web applications with Rust's safety and strong type checking.

1.4 Tools you'll need

Included as part of this book is a collection of code samples, freely available under the MIT license. To obtain a copy of the code, you will require an Internet-connected computer with a supported operating system and the following tools installed:

Table 1.2 Required tools

Name	Description
`git`	The source code for this book is stored in a public git repository, hosted on GitHub at https://github.com/brndnmtthws/code-like-a-pro-in-rust-book
`rustup`	Rust's tool for managing Rust components. `rustup` will manage your installation of `rustc` and other Rust components.
`gcc` or `clang`	You must have a copy of GCC or Clang installed to build certain code samples, but it's not required for most. Clang is likely the best choice for most people, and thus it's referred to by default. In cases where the `clang` command is specified, you may freely substitute `gcc` if you prefer.

For details on installing the tools above, refer to Appendix A.

1.5 Summary

- Rust is a modern system-level programming language with advanced safety features and zero-cost abstractions.
- Rust's steep learning curve can be an initial deterrent, but this book helps get over and past the hurdles.
- Rust has a lot of similarities to–and borrows concepts from–other languages, but it's quite unique, as explained throughout this book.
- Rust's vibrant community and mature package repository provide a rich ecosystem to build atop
- To get the most out of this book, follow along the code samples from https://github.com/brndnmtthws/code-like-a-pro-in-rust-book

2 Project management with Cargo

This chapter covers

- Introducing Cargo
- Managing Rust projects with Cargo
- Handling dependencies in Rust projects
- Linking to other (non-Rust) libraries
- Publishing Rust applications and libraries
- Documenting Rust code
- Following the Rust community's best practices for managing and publishing projects
- Structuring Rust projects with modules and workspaces
- Considerations for using Rust in embedded environments

Before we can jump into the Rust lanugage itself, we need to familiarize ourselves with the basic tools required to work with Rust. This may seem tedious, but I can assure you that mastering tooling is critical to success. The tools exist to make your life easier, so it will forever pay dividends to be an expert in the tools that were created by the language creators for the language users.

Rust's package management tool is called *Cargo*, and it's the interface to Rust's compiler `rustc`, the `https://crates.io` registry, and many other Rust tools (which we cover in more detail in chapter 3). Strictly speaking, it's possible to use Rust and `rustc` *without* using Cargo, but it's not something I'd recommend for most people.

When working with Rust, you'll likely spend a lot of time using Cargo and tools that work with Cargo. It's important to familiarize yourself with its use and best practices. In chapter 3, I'll provide recommendations and details on how to futher increase the usefulness of Cargo with community crates.

2.1 Cargo tour

To demonstrate Cargo's features, let's walk through a tour of Cargo and its typical usage. I implore you to follow along (ideally by running the commands as demonstrated below). In doing so, you may discover new features even if you're already familiar with Cargo and its usage.

2.1.1 Basic usage

To start, run `cargo help` to list available commands:

```
$ cargo help
Rust's package manager

USAGE:
    cargo [+toolchain] [OPTIONS] [SUBCOMMAND]

OPTIONS:
    -V, --version           Print version info and exit
        --list              List installed commands
        --explain <CODE>    Run `rustc --explain CODE`
    -v, --verbose           Use verbose output (-vv very verbose/build.rs
                            output)
    -q, --quiet             No output printed to stdout
        --color <WHEN>      Coloring: auto, always, never
        --frozen            Require Cargo.lock and cache are up to date
        --locked            Require Cargo.lock is up to date
        --offline           Run without accessing the network
    -Z <FLAG>...            Unstable (nightly-only) flags to Cargo, see
    'cargo -Z help' for details
    -h, --help              Prints help information

Some common cargo commands are (see all commands with --list):
    build, b    Compile the current package
    check, c    Analyze the current package and report errors, but don't
    build object files
    clean       Remove the target directory
    doc         Build this package's and its dependencies' documentation
    new         Create a new cargo package
    init        Create a new cargo package in an existing directory
    run, r      Run a binary or example of the local package
    test, t     Run the tests
    bench       Run the benchmarks
    update      Update dependencies listed in Cargo.lock
    search      Search registry for crates
    publish     Package and upload this package to the registry
    install     Install a Rust binary. Default location is $HOME/.cargo/bin
    uninstall   Uninstall a Rust binary

See 'cargo help <command>' for more information on a specific command.
```

If you run this yourself, your output may differ slightly depending on the version of Cargo you have installed. If you don't see output similar to the example above, you may need to verify your Cargo installation is working. Refer to Appendix A for details on installing Cargo.

2.1.2 Creating a new application or library

Cargo has a built in boilerplate generator, which can create a "Hello, world!" application or library, saving you time on getting started. To get started, run the following command in your shell from a development directory (I personally like to use `~/dev`):

```
$ cargo new dolphins-are-cool
     Created binary (application) `dolphins-are-cool` package
```

The command above creates a new boilerplate application called "dolphins-are-cool" (you can change the name to anything you want). Let's quickly examine the output:

```
$ cd dolphins-are-cool/
$ tree
.
 Cargo.toml
 src
     main.rs

1 directory, 2 files
```

Above, we see Cargo has created 2 files:

- `Cargo.toml` which is the Cargo configuration file for the new application, in *TOML* format
- `main.rs` inside the `src` directory, which represents the entry point for our new application

YOUR TURN **TOML, short for "Tom's Obvious, Minimal Language", is a configuration file format used by many Rust-related tools. For details on TOML, refer to https://toml.io.**

Next, use `cargo run` to compile and execute the newly created application:

```
$ cargo run
   Compiling dolphins-are-cool v0.1.0 (/Users/brenden/dev/dolphins-are-cool)
    Finished dev [unoptimized + debuginfo] target(s) in 0.59s
     Running `target/debug/dolphins-are-cool`
Hello, world!   ❶
```

❶ This is the Rust program output.

Running the `cargo new` command as above, but with the `--lib` argument, will create a new library:

```
$ cargo new narwhals-are-real --lib
     Created library `narwhals-are-real` package
$ cd narwhals-are-real/
$ tree
.
 Cargo.toml
 src
     lib.rs

1 directory, 2 files
```

The code generated from `cargo new --lib` is slightly different, as it contains a single unit test in `src/lib.rs` rather than a `main` function. You can run the tests with `cargo test`:

```
$ cargo test
    Finished test [unoptimized + debuginfo] target(s) in 0.00s
     Running target/debug/deps/narwhals_are_real-3265ca33d2780ea2

running 1 test
test tests::it_works ... ok

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

   Doc-tests narwhals-are-real

running 0 tests

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

YOUR TURN **Applications use `src/main.rs` as their entrypoint, and libraries use `src/lib.rs` as their entrypoint.**

When using `cargo new`, Cargo will automatically initialize the new directory as a Git repository (*except* when already inside a repository), including a `.gitignore` file. Cargo also supports hg, pijul, or fossil with the `--vcs` flag.

2.1.3 Building, running, and testing

The Cargo commands you'll likely spend the most time working with are `build`, `check`, `test`, and `run`. They are summarized as follows:

Table 2.1 Cargo build & run commands

Cargo Command	Summary
`build`	compiles and links your package, creating all final targets
`check`	similar to build except does not actually generate any targets or objects, merely checks the validity of code
`test`	compiles and runs all tests
`run`	compiles and runs the target binary

The commands you will likely spend a great deal of time working with are `cargo check` and `cargo test`. By using `check`, you can save time and iterate quickly while writing code as it will validate syntax faster than `cargo build`. To illustrate this, let's time the compilation of the *dryoc* crate, available from https://github.com/brndnmtthws/dryoc, which I use throughout this book for example purposes:

```
$ cargo clean
$ time cargo build
...
    Finished dev [unoptimized + debuginfo] target(s) in 9.26s
cargo build  26.95s user 5.18s system 342% cpu 9.374 total
$ cargo clean
$ time cargo check
...
    Finished dev [unoptimized + debuginfo] target(s) in 7.97s
cargo check  23.24s user 3.80s system 334% cpu 8.077 total
```

In this particular case, the difference is not substantial: about 9.374 seconds for the `build` command versus 8.077 seconds for `check` (according to the wall-clock time given by the `time` command). However, on larger crates the time saved can become substantial. Additionally, there's a multiplicative effect as you often recompile (or recheck) the code over and over when iterating on changes.

2.1.4 Switching between toolchains

A *toolchain* is a combination of architecture, platform, and channel. One example is `stable-x86_64-apple-darwin`, which is the stable channel for x64-64 Darwin (equivalent to Apple's macOS on Intel CPUs). Rust is published as 3 different *channels*: stable, beta, and nightly. Stable is the least frequently updated, most well tested channel. Beta contains features that are ready for stabilization, but require further testing and are subject to change. Nightly contains unreleased language features that are considered a work in progress.

When working with Rust, you'll often find yourself needing to switch between different toolchains. In particular, you may often need to switch between the stable and nightly channels. An easy way to do this directly with Cargo is to use the `+channel` option, like so:

```
# Runs tests with stable channel:
$ cargo +stable test
...
# Runs tests with nightly channel:
$ cargo +nightly test
...
```

NOTE **You may need to install the nightly toolchain with `rustup toolchain install nightly` before running any `cargo +nightly …` commands, if you haven't already done so.**

This option works with all Cargo commands, and it's the quickest way to switch between toolchains. The alternative is to switch your default toolchain using `rustup`, covered in Appendix A.

In many cases, you'll want to test your code with both stable and nightly before publishing, especially with open source projects as many people use both toolchains. Additionally, many

Rust projects are nightly *only*, which is discussed in more detail in chapter 3.

You may also use the *override* option with `rustup`, which allows you to set the toolchain for a specific project or directory. For example, you can set the current working directory to the `nightly` channel with:

```
# Only applies to the current directory and its children
$ rustup override set nightly
```

This is quite handy, as it allows you to keep the stable channel by default, but switch to nightly for specific projects.

2.2 Dependency management

92,706 The `crates.io` package (or *crate*) registry is one of Rust's force multipliers. In the Rust community, packages are called *crates* and they include both applications and libraries. As of the time of writing, there are more than 92,000[2] different crates available.

Most of the time throughout this book when referring to crates, we're likely to be using libraries rather than applications. In chapter 3 we'll discuss more Rust tooling that can be installed from crates, but most of the time you'll be using libraries.

Rust has a unique approach compared to some programming languages, in that the core language itself does not include many features. By comparison, languages like Java, C#, and even C++ to some degree include significant components as part of the core language (either in the runtime, or as part of the compiler). For example, Rust's core data structures–compared to other languages–are quite minimal, and many are just wrappers around the core data resizable structure, `Vec`. Rust prefers to provide features through crates, rather than creating a large standard library.

The Rust language itself doesn't even include random number generator, which is critical for a lot of programming tasks. For that you need to use the `rand` crate, which is the most downloaded crate as of the time of writing (or write your own random number generator).

If you're coming from a language like JavaScript, Ruby, or Python then Rust's crates will be somewhat familiar to their corresponding package management tools. Compared to languages like C, C++ it's like discovering fire for the first time. Gone are the days of manually writing complicated build checks for 3rd party libraries, or integrating 3rd party code and build systems into your own source repository.

Describing dependencies in Rust is done by listing them in `Cargo.toml`. A simple example using the `rand` crate looks like so:

Listing 2.1 A minimal `Cargo.toml`

```
[package]
name = "simple-project"
version = "0.1.0"
authors = ["Brenden Matthews <brenden@brndn.io>"]
edition = "2018"

[dependencies]
rand = "0.8"
```

In the example above, we're including the `rand` crate, using the latest 0.8 release of the library. When specifying dependency versions, you should follow Semantic Versioning[3], which uses the `major.minor.patch` pattern. By default, Cargo will use *caret* requirements if an operator is not specified, which permits updates to the least specified version.

You can also add a dependency to a project with the `cargo add` command:

```
# Adds the rand crate as a dependency to the current project
$ cargo add rand
```

Cargo supports caret (`^x.y.z`), tilde (`~x.y.z`), wildcard (`*`, `x.*`), comparison requirements (`>=x`, `<x.y`, `=x.y.z`), and combinations thereof. In practice, you would specify the library version as `major.minor` (allows compatible upgrades under the caret rules), or `=major.minor.patch` (pinned to a specific version). Refer to https://doc.rust-lang.org/cargo/reference/specifying-dependencies.html for reference on dependency specifications.

Table 2.2 Summary of SemVer dependency specification

Operator	Example	Min version	Max version	Updates?
Caret	^2.3.4	>=2.3.4	<3.0.0	Allowed
Caret	^2.3	>=2.3.0	<3.0.0	Allowed
Caret	^0.2.3	>=0.2.3	<0.3.0	Allowed
Caret	^2	>=2.0.0	<3.0.0	Allowed
Tilde	~2.3.4	>=2.3.4	<2.4.0	Allowed
Tilde	~2.3	>=2.3.0	<2.4.0	Allowed
Tilde	~0.2	>=0.2.0	<0.3	Allowed
Wildcard	2.3.*	>=2.3.0	<2.4.0	Allowed
Wildcard	2.*	>=2.0.0	<3.0.0	Allowed
Wildcard	*	None	None	Allowed
Comparison	=2.3.4	=2.3.4	=2.3.4	No
Comparison	>=2.3.4	>=2.3.4	None	Allowed
Comparison	>=2.3.4,<3.0.0	>=2.3.4	<3.0.0	Allowed

Internally, Cargo uses the `semver` crate[4] for parsing the versions specified. When you run `cargo`

`update` within your project, Cargo will update the `Cargo.lock` file with the newest available crates as per your dependency specification.

> **YOUR TURN** **Avoid pinning dependency versions, especially in libraries. It can cause headaches down the road when downstream packages need different versions of common libraries.**

How exactly to specify dependencies is a topic of much debate. There are no hard and fast rules, but you should generally assume other projects follow SemVer. Some projects adhere to SemVer rules strictly, and others do not. In most cases it needs to be evaluated on a case-by-case basis. A reasonable default assumption is to allow upgrades to minor and patch versions by specifying the minimum required version with the caret operator, which is the default in Rust (if you don't explicitly specify an operator).

For your own published crates, please do follow SemVer, as it helps other developers build on your work and preserve compatibility.

2.2.1 Handling the *`Cargo.lock`* file

Handling `Cargo.lock` requires a bit of special consideration, at least with regard to version control systems. The file contains a list of the package dependencies (both direct and indirect dependencies), their versions, and checksums for verifying integrity.

If you're coming from languages with similar package management systems, you've probably seen similar files before (npm uses `package-lock.json`, Ruby gems use `Gemfile.lock`, Python Poetry uses `poetry.lock`). For libraries, it's recommended you *do not* include this file in your version control system. When using Git, you can do this by adding `Cargo.lock` to `.gitignore`. Leaving out the lock file allows downstream packages to update indirect dependencies as needed.

For applications, it's recommended you always include `Cargo.lock` alongside `Cargo.toml`. This helps to ensure consistent behaviour in published releases, should 3rd party libraries change in the future.

This is a well-established convention, and not unique to Rust. Lastly, Cargo will automatically create an appropriate `.gitignore` file for you and initialize a Git repository.

2.3 Feature flags

It's common practice when publishing software, particularly libraries, to have optional dependencies. This is usually for the purpose of keeping compile times low, binaries small, and perhaps providing performance improvements, with the trade off of some additional complexity at compile time.

In some cases you may want to include optional dependencies as part of your crate. These can be expressed as feature flags with Cargo. There are some limitations with feature flags, notably they only permit boolean expressions (i.e., enabled or disabled). Feature flags are also passed through to crates in your dependency list, so you can enable features for underlying crates through top level feature flags.

I recommend not relying too heavily on feature flags. You may find yourself leaning toward creating supercrates with lots of feature flags, but if you find yourself doing this you may want to instead break your crate into smaller separate subcrates. This pattern is quite common, some good examples are the `serde`, `rand`, and `rocket` crates. There are some cases where you *must* use feature flags to express certain optional features, such as when providing optional trait implementations in the top level crate.

To examine how feature flags are used in practice, let's look at the `dryoc` crate, which uses a few flags to express some features: `serde`, `base64` for binary encoding (with `serde`), and SIMD optimizations.

Listing 2.2 Code listing for `Cargo.toml` from `dryoc` crate:

```
[dependencies]
base64 = {version = "0.13", optional = true}   ❶
curve25519-dalek = "3.0"
generic-array = "0.14"
poly1305 = "0.6"
rand_core = {version = "0.5", features = ["getrandom"]}
salsa20 = {version = "0.7", features = ["hsalsa20"]}
serde = {version = "1.0", optional = true, features = ["derive"]}   ❷
sha2 = "0.9"
subtle = "2.4"
x25519-dalek = "1.1"
zeroize = "1.2"

[dev-dependencies]
base64 = "0.13"
serde_json = "1.0"
sodiumoxide = "0.2"

[features]   ❸
default = [   ❹
  "u64_backend",
]
simd_backend = ["curve25519-dalek/simd_backend", "sha2/asm"]   ❺
u32_backend = ["x25519-dalek/u32_backend"]
u64_backend = ["x25519-dalek/u64_backend"]
```

❶ Optional base64 dependency, is not included by default.

❷ Optional serde dependency, not included by default.

❸ Default and optional features section.

❹ The list of default features.

❺ Optional features, and the features they switch on for dependencies.

Table 2.3 Summary of SemVer dependency specification

Flag	Description	Enabled by default?
serde	Enables optional `serde` dependency	No
base64	Enables `base64` dependency, but only activates when `serde` also enabled (see below for details)	No
simd_backend	Enables the SIMD and asm features for `curve25519-dalek` and `sha2` crates	No
u64_backend	Enables the u64 backend for the `x25519-dalek` crate, which is mutually exclusive with `u32_backend`	Yes
u32_backend	Enables the u32 backend for the `x25519-dalek` crate, which is mutually exclusive with the `u64_backend`	No

Next, let's examine some of the crate's code to see how these flags are used by utilizing `cfg` and `cfg_attr`, which instruct the Rust compiler, `rustc`, how to use these flags. We'll look at `src/message.rs`, which demonstrates the use of feature flags.

Listing 2.3 Partial code listing for `src/message.rs` from `dryoc` crate:

```
//
#[cfg(feature = "serde")]   ❶
use serde::{Deserialize, Serialize};

use zeroize::Zeroize;

#[cfg_attr(   ❷
    feature = "serde",   ❷
    derive(Serialize, Deserialize, Zeroize, Debug, PartialEq)   ❷
)] // B
#[cfg_attr(not(feature = "serde"), derive(Zeroize, Debug, PartialEq))]   ❸
#[zeroize(drop)]
/// Message container, for use with unencrypted messages
pub struct Message(pub Box<InputBase>);
```

❶ Enables `use` statement only when serde is enabled.

❷ Enables the `derive()` statement only when serde is enabled.

❸ Enables the `derive()` statement only when serde is disabled.

The code above uses several conditional compilation attributes:

- **cfg(*predicate*)**: instructs the compiler to only compile the thing its attached to if the predicate is true

- **cfg_attr(*predicate, attribute*)**: instructs the compiler to only enable the specified attribute (second argument) if the predicate (first argument) is true
- **not(*predicate*)**: returns true if the predicate is false and vice versa

Additionally, you may use `all(predicated)` and `any(predicate)` which return true when all or any of the predicates are true respectively.

For more examples, see `src/lib.rs`, `src/b64.rs`, and `src/dryocbox.rs` and `src/dryocsecretbox.rs` within the `dryoc` crate.

> **YOUR TURN** **When you generate documentation for a project with `rustdoc`, it automatically provides a feature flag listing for you. We'll explore `rustdoc` in detail later in this chapter.**

2.4 Patching dependencies

One problem you may encounter from time to time is the need to patch an upstream crate (i.e., a crate you depend on from *outside* your project). I have encountered many instances where I need to update another crate I depend on, usually for some minor issue. It's rarely worth the trouble of replacing functionality of upstream crates just to fix one or two minor issues. In some cases, you may be able to simply switch to the pre-release version of the crate, or else you have to patch it yourself.

The process for patching an upstream crate goes something like this:

1. Create a fork on GitHub
2. Patch the crate in your fork
3. Submit a pull request to the upstream project
4. Change your `Cargo.toml` to point to your fork while waiting for the pull request to be merged and released

This process is not without problems. One issue is keeping track of changes to the upstream crate and integrating them as needed. Another problem is that your patch may never be accepted upstream, in which case you can get stuck on a fork. When working with upstream crates, you should try to avoid forking when possible.

Cargo provides a way for you to patch crates using the fork method above without too much fuss, however there are some caveats. To illustrate, let's walk through the typical process for patching a crate. For this example, I'll make a local copy of the source code rather than creating a forked project on GitHub.

Let's modify the `num_cpus` crate, to replace it with our own patched version (I chose this crate for its simplicity; it returns the number of logical CPU cores).

Start by creating an empty project:

```
$ cargo new patch-num-cpus
...
$ cd patch-num-cpus
$ cargo run
...
Hello, world!
```

Next, add the `num_cpus` dependency to `Cargo.toml`:

```
[dependencies]
num_cpus = "1.0"
```

Update `src/main.rs` to print the number of CPUs:

```
fn main() {
    println!("There are {} CPUs", num_cpus::get());
}
```

Finally, run the new crate:

```
$ cargo run
    Finished dev [unoptimized + debuginfo] target(s) in 0.00s
     Running `target/debug/patch-num-cpus`
There are 4 CPUs
```

At this point we haven't patched or modified anything. Let's create a new library within the same working directory, where we'll just reimplement the same API:

```
$ cargo new num_cpus --lib
...
```

Next, we'll patch the default `src/lib.rs` to implement `num_cpus::get()`. Update `src/lib.rs` from the `num_cpus` directory to look like this:

```
pub fn get() -> usize {
    100 // Return some arbitrary value, for test purposes
}
```

Now we have `num_cpus` with our own implementation that returns a rather pointless hard-coded value (100, in this case). Go back up a directory to the original `patch-num-cpus` project, and modify `Cargo.toml` to use the replacement crate:

```
[dependencies]
num_cpus = { path = "num_cpus" }
```

Run the same code with the patched crate:

```
$ cargo run
   Compiling patch-num-cpus v0.1.0
    Finished dev [unoptimized + debuginfo] target(s) in 0.33s
     Running `target/debug/patch-num-cpus`
There are 100 CPUs
```

This example is fairly pointless, but it effectively illustrates the process. If you want to patch a dependency using a fork from GitHub, for example, you would point your dependency directly to your GitHub repository, like this (in `Cargo.toml`):

```
[dependencies]
num_cpus = { git = "https://github.com/brndnmtthws/num_cpus",
  rev = "b423db0a698b035914ae1fd6b7ce5d2a4e727b46" }
```

If you execute `cargo run` now, you should again see the correct number of CPUs reported (as I created the fork above, but without any changes). `rev` in the example above is referring to a Git hash for the latest commit at the time of writing. When you compile the project, Cargo will fetch the source code from the GitHub repository, checkout the particular revision specified (could be a commit, branch, or tag), and compile that version as a dependency.

2.4.1 Indirect dependencies

Sometimes you need to patch dependencies of dependencies. That is to say, you might depend on a crate which depends on another crate that requires patching. Using `num_cpus` as an example, the crate currently depends on `libc = "0.2.26"` (but only on non-Windows platforms). For the sake of this example, we can patch that dependency to a newer release, by updating `Cargo.toml` like so:

```
[patch.crates-io]
libc = { git = "https://github.com/rust-lang/libc", tag = "0.2.88" }
```

Above, we're going to point to the Git repository for `libc` and specify the `0.2.88` tag explicitly. The `patch` section in `Cargo.toml` serves as a way to patch the `crates.io` registry itself, rather than patching a package directly. You are, in effect, replacing all upstream dependencies for `libc` with your own version.

Use this feature carefully, and only under special circumstances. It does not affect downstream dependencies, meaning any crates that depend on your crate won't inherit the patch. This is a limitation of Cargo that currently does not have a reasonable workaround.

In cases where you need more control over 2nd and 3rd order dependencies, you'll need to either fork all the projects involved or include them directly in your own project as subprojects using workspaces (discussed later in this chapter).

2.4.2 Best practices with dependency patching

- Avoid patching dependencies, as it can be difficult to maintain over time. When patching is necessary, submit patches upstream with the required patches.
- Avoid forking upstream crates, and in cases where it's unavoidable, try to get back onto the main branch as quickly as possible.

2.5 Publishing crates

For projects you wish to publish to `crates.io`, the process is simple. Once your crate is ready to go, you can run `cargo publish` and Cargo takes care of the details. There are a few requirements for publishing a crate, such as specifying a license, providing certain project details like documentation and a repository URL, and ensuring all dependencies are also available to `crates.io`.

It is possible to publish to a private registry, however Cargo's support at the time of writing for private registries is quite limited. Thus, it's recommended you use private Git repositories and tags instead of relying on `crates.io` for private crates.

2.5.1 CI/CD integration

For most crates, you'll want to set up a system for publishing releases to `crates.io` automatically. CI/CD (continuous integration, continuous deployment) systems are a common component on modern development cycles. They're usually composed of 2 distinct steps:

- **Continuous Integration (CI)**: a system that compiles, checks, and verifies each commit to a VCS repository
- **Continuous Deployment (CD)**: a system that automatically deploys each commit or release, provided it passes all necessary checks from the CI

To demonstrate this, I will walk through the `dryoc` project, which uses GitHub Actions[5] (which is freely available for open source projects).

Before looking at the code, let's describe the release process with a typical Git workflow once you've decided it's time to publish a release:

1. If needed, update the `version` attribute within `Cargo.toml` to the version you want to release.
2. The continuous integration system will run, verifying all the tests and checks pass.
3. You'll create and push a tag for the release (use a version prefix, i.e., `git tag -s vX.Y.Z`).
4. The continuous deployment system will run, build the tagged release, and publish to `crates.io` with `cargo publish`.
5. Update the `version` attribute in `Cargo.toml` for the *next* release cycle in a new commit.

Let's examine the `dryoc` crate, which implements this pattern using GitHub Actions. There are 2 separate actions to look at:

- `.github/workflows/build-and-test.yml` [6]: builds and runs tests for a combination of features, platforms, and toolkits
- `.github/workflows/publish.yml` [7]: builds and runs tests for a tagged release matching the `v*` pattern, publishing the crate to `crates.io`

Listing 2.4 shows the build job paramaters, including the feature, channel, and platform matrix.

Listing 2.4 Partial code listing for `.github/workflows/build-and-test.yml`:

```
name: Build & test
on:   ❶
  push:
    branches: [main]
  pull_request:
    branches: [main]
env:
  CARGO_TERM_COLOR: always
jobs:
  build:
    strategy:
      matrix:
        rust:   ❷
          - stable
          - beta
          - nightly
        features:   ❸
          - serde
          - serde,base64
          - simd_backend
          - default
        os:   ❹
          - ubuntu-18.04
          - ubuntu-20.04
          - macos-10.15
          - windows-2019
        exclude:   ❺
          - rust: stable
            features: simd_backend
          - rust: beta
            features: simd_backend
          - os: windows-2019
            features: simd_backend
```

❶ Builds will only run on Git pushes and pull requests for the `main` branch.

❷ Runs with stable, beta, and nightly channels.

❸ Runs tests with different features enabled separately.

❹ Runs on Linux, macOS, and Windows.

❺ Some build combinations don’t work, so they’re disabled here.

Listing 2.5 shows the individual steps to build, test, format, and run clippy.

Listing 2.5 Partial code listing for `.github/workflows/build-and-test.yml`:

```
runs-on: ${{ matrix.os }}
steps:
  - uses: actions/checkout@v2
  - name: Install Rust toolchain   ❶
    uses: actions-rs/toolchain@v1
    with:
      profile: minimal
      toolchain: ${{ matrix.rust }}
      override: true
      components: rustfmt, clippy   ❷
  - name: Build   ❸
    uses: actions-rs/cargo@v1
    with:
      command: build
      args: --features ${{ matrix.features }}
  - name: Test   ❹
    uses: actions-rs/cargo@v1
    with:
      command: test
      args: --features ${{ matrix.features }}
  - name: Rustfmt   ❺
    uses: actions-rs/cargo@v1
    with:
      command: fmt
      args: --all -- --check
  - name: Clippy   ❻
    uses: actions-rs/cargo@v1
    with:
      command: clippy
      args: --features ${{ matrix.features }} -- -D warnings
```

❶ This step installs the desired toolchain.

❷ Installs additional Cargo components.

❸ Runs build with the specified features.

❹ Runs all tests with the specified features.

❺ Verifies code formatting (rustfmt is discussed in chapter 3).

❻ Runs Clippy checks with the specified features (Clippy is discussed in chapter 3).

In listing 2.6, we show the steps involved to publish our crate.

Listing 2.6 Code listing for `.github/workflows/publish.yml`

```
name: Publish to crates.io
on:
  push:
    tags:
      - v*   ❶
env:
  CARGO_TERM_COLOR: always
jobs:
  build:
    runs-on: ubuntu-20.04
    steps:
      - uses: actions/checkout@v2
      - name: Install Rust toolchain
        uses: actions-rs/toolchain@v1
        with:
          profile: minimal
          toolchain: stable
          override: true
      - name: Build
        uses: actions-rs/cargo@v1
        with:
          command: build
      - name: Test
        uses: actions-rs/cargo@v1
        with:
          command: test
      - name: Login to crates.io   ❷
        uses: actions-rs/cargo@v1
        with:
          command: login
          args: -- ${{ secrets.CRATES_IO_TOKEN }}   ❸
      - name: Publish to crates.io   ❹
        uses: actions-rs/cargo@v1
        with:
          command: publish
```

❶ Only runs when tag matches `v*`.

❷ Logs in using secret stored in repository's secrets configuration.

❸ This token is stored using GitHub's secret storage feature, which must be supplied ahead of time.

❹ Publishes crate to `https://crates.io`.

NOTE **GitHub's Actions doesn't currently support any way to gate a release when using separate stages (i.e., wait until the build stages succeed before proceeding with the deploy stage). To accomplish this, you must verify the build stage succeeds before pushing any tags.**

In the final publish step, you'll need to provide a token for `https://crates.io`. This can be done by creating a `crates.io` account, generating a token from the `crates.io` account settings, and adding it to GitHub's secret storage in the settings for your GitHub repository.

2.6 Linking to C libraries

You may occasionally find yourself needing to use external libraries from non-Rust code. This is usually accomplished with FFI (foreign function interface). FFI is a fairly standard way to accomplish cross-language interoperability. We'll revisit FFI again in more detail in chapter 4.

Let's walk through a simple example of calling functions from one of the most popular C libraries: zlib. Zlib was chosen because it's nearly ubiquitous, and this example should work easily out of the box on any platform where zlib is available. We'll implement 2 functions in Rust, `compress()` and `uncompress()`. Here are the definitions from the zlib library (which has been simplified for the purposes of this example):

Listing 2.7 Simplified code listing from `zlib.h`

```
int compress(void *dest, unsigned long *destLen,
             const void *source, unsigned long sourceLen);

unsigned long compressBound(unsigned long sourceLen);

int uncompress(void *dest, unsigned long *destLen,
               const void *source, unsigned long sourceLen);
```

First, we'll define the C interface in Rust using `extern`:

Listing 2.8 Code listing for zlib utility functions

```
use libc::{c_int, c_ulong};

#[link(name = "z")]
extern "C" {
    fn compress(
        dest: *mut u8,
        dest_len: *mut c_ulong,
        source: *const u8,
        source_len: c_ulong,
    ) -> c_int;
    fn compressBound(source_len: c_ulong) -> c_ulong;
    fn uncompress(
        dest: *mut u8,
        dest_len: *mut c_ulong,
        source: *const u8,
        source_len: c_ulong,
    ) -> c_int;
}
```

We've included `libc` as a dependency, which provides C-compatible types in Rust. Whenever you're linking to C libraries, you'll want to use types from `libc` to maintain compatibility. Failure to do so may result in undefined behaviour. We've defined 3 utility functions from zlib: `compress`, `compressBound`, and `uncompress`.

The `link` attribute tells `rustc` that we need to link these functions to the `z` library. This is equivalent to adding the `-lz` flag at link time. On macOS, you can verify this with `otool -L` (on Linux use `ldd`, on Windows use `dumpbin`):

```
$ otool -L target/debug/zlib-wrapper
target/debug/zlib-wrapper:
    /usr/lib/libz.1.dylib (compatibility version 1.0.0, current version
  1.2.11)
    /usr/lib/libiconv.2.dylib (compatibility version 7.0.0, current version
  7.0.0)
    /usr/lib/libSystem.B.dylib (compatibility version 1.0.0, current version
  1292.60.1)
    /usr/lib/libresolv.9.dylib (compatibility version 1.0.0, current version
  1.0.0)
```

Next, we need to write Rust functions that wrap the C functions and can be called from Rust code. Calling C functions directly is considered unsafe in Rust, so you must wrap it in an `unsafe {}` block:

Listing 2.9 Code listing for `zlib_compress`

```
pub fn zlib_compress(source: &[u8]) -> Vec<u8> {
    unsafe {
        let source_len = source.len() as c_ulong;

        let mut dest_len = compressBound(source_len);   ❶
        let mut dest = Vec::with_capacity(dest_len as usize);   ❷

        compress(   ❸
            dest.as_mut_ptr(),
            &mut dest_len,
            source.as_ptr(),
            source_len,
        );
        dest.set_len(dest_len as usize);
        dest   ❹
    }
}
```

❶ Returns the upper bound of the length of the compressed output.

❷ Allocates dest_len bytes on heap using a `Vec`.

❸ Calls zlib C function.

❹ Returns the result as a `Vec`.

The `zlib_uncompress` version of the function above is nearly identical, except we need to provide our own length for the destination buffer. Finally, we can demonstrate the usage as such:

Listing 2.10 Code listing for `main()`

```
fn main() {
    let hello_world = "Hello, world!".as_bytes();
    let hello_world_compressed = zlib_compress(&hello_world);
    let hello_world_uncompressed =
        zlib_uncompress(&hello_world_compressed, 100);
    assert_eq!(hello_world, hello_world_uncompressed);
    println!(
        "{}",
        String::from_utf8(hello_world_uncompressed)
            .expect("Invalid characters")
    );
}
```

The biggest challenge when dealing with FFI is the complexity of some C APIs, and mapping the various types and functions. To work around this, you can use the rust `bindgen` tool, which is discussed in more detail in chapter 4.

2.7 Binary distribution

Rust's binaries are composed of all Rust dependencies for a given platform, included as a single binary, excluding the C runtime in addition to any non-Rust libraries that may have been dynamically linked. You *can* build binaries that are statically linked to the C runtime, but by default this is optional. Thus, when distributing Rust binaries, you'll need to consider whether you want to statically link the C runtime, or rely on the system's runtime.

The binaries themselves *are* platform dependent. They can be cross-compiled for different platforms, but you cannot mix different architectures or platforms with the same Rust binary. A binary compiled for Intel-based x64-64 CPUs will not run on ARM-based platforms like aarch64 (also known as armv8), without some type of emulation. A binary compiled for macOS won't run on Linux.

Some OS vendors, notably Apple's macOS, provide emulation for other CPU platforms. It's possible to run x86-64 binaries automatically on ARM using Apple's Rosetta tool, which should happen automatically. For more detail on macOS binary distribution, you'll need to consult Apple's developer documentation at https://developer.apple.com/documentation/apple-silicon/building-a-universal-macos-binary.

In most cases, you'll want to stick with the defaults for the platform you're using, but there are some exceptions to where you don't want to do this.

If you're coming from a language such as Go, you may have become accustomed to distributing pre-compiled binaries without worrying about the C runtime. Unlike Go, Rust requires a C runtime, and it uses dynamic linking by default.

2.7.1 Cross compilation

You can use Cargo to cross-compile binaries for different targets, but only where compiler support is available for that target. For example, you can easily compile Linux binaries on Windows, but compiling Windows binaries on Linux is not as easy (but not impossible).

You can list the available targets on your host platform using `rustup`:

```
$ rustup target list
rustup target list
aarch64-apple-darwin
aarch64-apple-ios
aarch64-fuchsia
aarch64-linux-android
aarch64-pc-windows-msvc
..
```

You can install different targets with `rustup target add <target>`, and then use `cargo build --target <target>` to build for a particular target.

For example, on my Intel-based macOS machine, I can run the following to compile binaries for aarch64 (used by the M1 chip):

```
$ rustup target add aarch64-apple-darwin
info: downloading component 'rust-std' for 'aarch64-apple-darwin'
info: installing component 'rust-std' for 'aarch64-apple-darwin'
info: using up to 500.0 MiB of RAM to unpack components
 18.3 MiB /  18.3 MiB (100 %)  14.7 MiB/s in  1s ETA:  0s
$ cargo build --target aarch64-apple-darwin
...
    Finished dev [unoptimized + debuginfo] target(s) in 3.74s
```

However, if I try to run the binary it will fail:

```
$ ./target/aarch64-apple-darwin/debug/simple-project
-bash: ./target/aarch64-apple-darwin/debug/simple-project: Bad CPU type in
executable
```

If I had access to an aarch64 macOS device, I could copy this binary to that machine and run it there successfully.

2.7.2 Building statically linked binaries

Normal Rust binaries include all the compiled dependencies, *except* for the C runtime library. On Windows and macOS, it's normal to distribute pre-compiled binaries and link to the OS's C runtime libraries. On Linux, however, most packages are compiled from source by the distribution's maintainers, and the distributions take responsibility for managing the C runtime.

When distributing Rust binaries on Linux, you can use *either* glibc or musl, depending on your preference. glibc is the default C library runtime on most Linux distributions. However, I recommend statically linking to musl when you want to distribute Linux binaries for maximum portability.

In fact, when trying to statically link on certain targets, Rust assumes you want to use musl.

NOTE **musl behaves slightly differently from glibc in certain cases, which is documented on the musl wiki at https://wiki.musl-libc.org/functional-differences-from-glibc.html.**

You can instruct `rustc` to use a static C runtime with the `target-feature` flag, like this:

```
$ RUSTFLAGS="-C target-feature=+crt-static" cargo build
    Finished dev [unoptimized + debuginfo] target(s) in 0.01s
```

Above, we're passing `-C target-feature=+crt-static` to `rustc` via the `RUSTFLAGS` environment variable which is interpreted by Cargo and passed to `rustc`.

To link statically to musl on x86-64 Linux:

```
# Make sure musl target is installed
$ rustup target add x86_64-unknown-linux-musl
...
# Compile using musl target and force static C runtime
$ RUSTFLAGS="-C target-feature=+crt-static" cargo build --target
x86_64-unknown-linux-musl
...
```

To explicitly disable static linking, use `RUSTFLAGS="-C target-feature=-crt-static"` instead (by flipping the plus `+` to minus `-`). This may be desirable on targets that default to static linking–if unusure, use the default parameters.

Alternatively, you can specify `rustc` flags for Cargo with `~/.cargo/config`:

```
[target.x86_64-pc-windows-msvc]
rustflags = ["-Ctarget-feature=+crt-static"]
```

The example above, when added to `~/.cargo/config`, will instruct `rustc` to link statically when using the `x86_64-pc-windows-msvc` target.

2.8 Documenting Rust projects

Rust's tool for documenting code is called `rustdoc`, which ships with Rust by default. If you've used code documentation tools from other projects (such as Javadoc, Docstring, rdoc, etc), `rustdoc` will come naturally.

Using `rustdoc` is as simple as adding comments in your code, and generating docs. Let's run through a quick example. Start by creating a library:

```
$ cargo new rustdoc-example --lib
     Created library `rustdoc-example` package
```

Now let's edit `src/lib.rs` to add a function called `mult` which takes two integers (a and b) and multiplies them. We'll also add a test:

```
pub fn mult(a: i32, b: i32) -> i32 {
    a * b
}

#[cfg(test)]
mod tests {
    use super::*;
    #[test]
    fn it_works() {
        assert_eq!(2 * 2, mult(2, 2));
    }
}
```

We haven't added any documentation yet. Before we do, let's generate some empty documentation using Cargo:

```
$ cargo doc
 Documenting rustdoc-example v0.1.0
 (/Users/brenden/dev/code-like-a-pro-in-rust/code/c2/2.8/rustdoc-example)
    Finished dev [unoptimized + debuginfo] target(s) in 0.89s
```

Now, you should see the generated HTML docs in `target/`. If you want to open the docs in a browser, you can open `target/doc/src/rustdoc_example/lib.rs.html` to view them. The result should look like this:

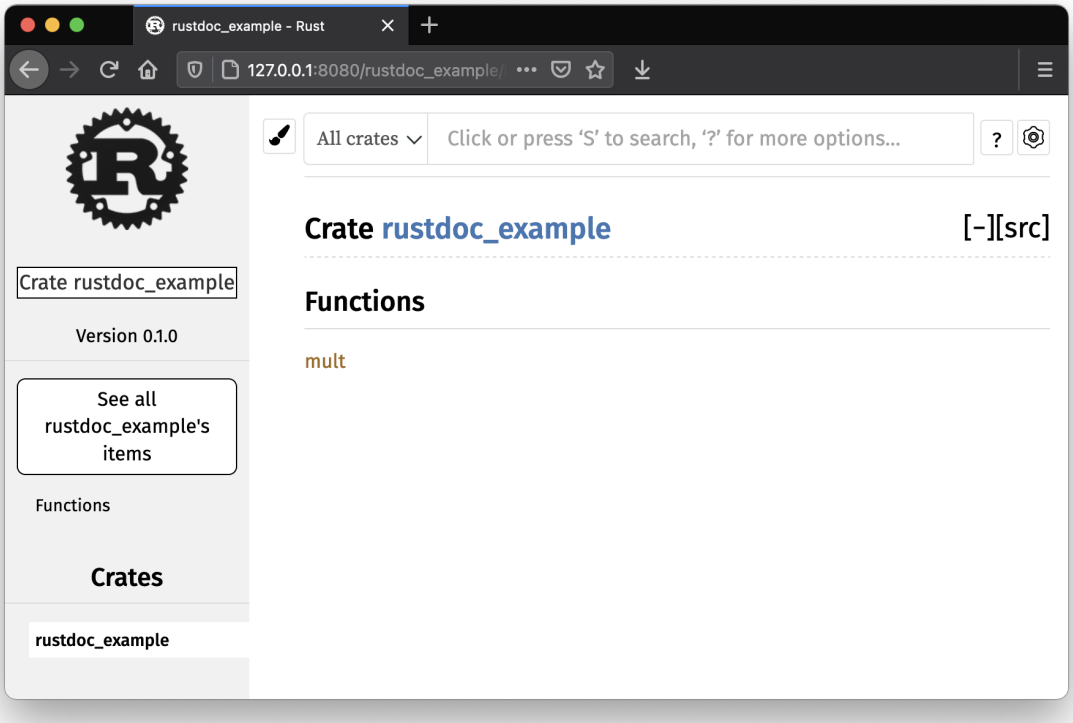

Figure 2.1 Screenshot of empty `rustdoc` HTML output

The default docs are empty, but you can see the public function `mult` listed in the docs.

Next, let's add a compiler attribute and some docs to our project. Update `src/lib.rs` so that it looks like this:

```
// This is a crate-level doc string, which appears on the front page for
// the crate's docs.
//! # rustdoc-example
//!
//! A simple project demonstrating the use of rustdoc with the function
//! [`mult`].

#![warn(missing_docs)]   ❶

/// Returns the product of `a` and `b`.   ❷
pub fn mult(a: i32, b: i32) -> i32 {
    a * b
}
```

❶ This compiler attribute tells `rustc` to generate a warning when docs are missing for public functions, modules, or types.

❷ This comment provides the documentation for the function `mult`.

YOUR TURN **Rust documentation is formatted using commonmark, a subset of Markdown. A reference for commonmark can be found at https://commonmark.org/help .**

If you re-run `cargo doc` with the newly created code documentation and open it in a browser, you will see the following:

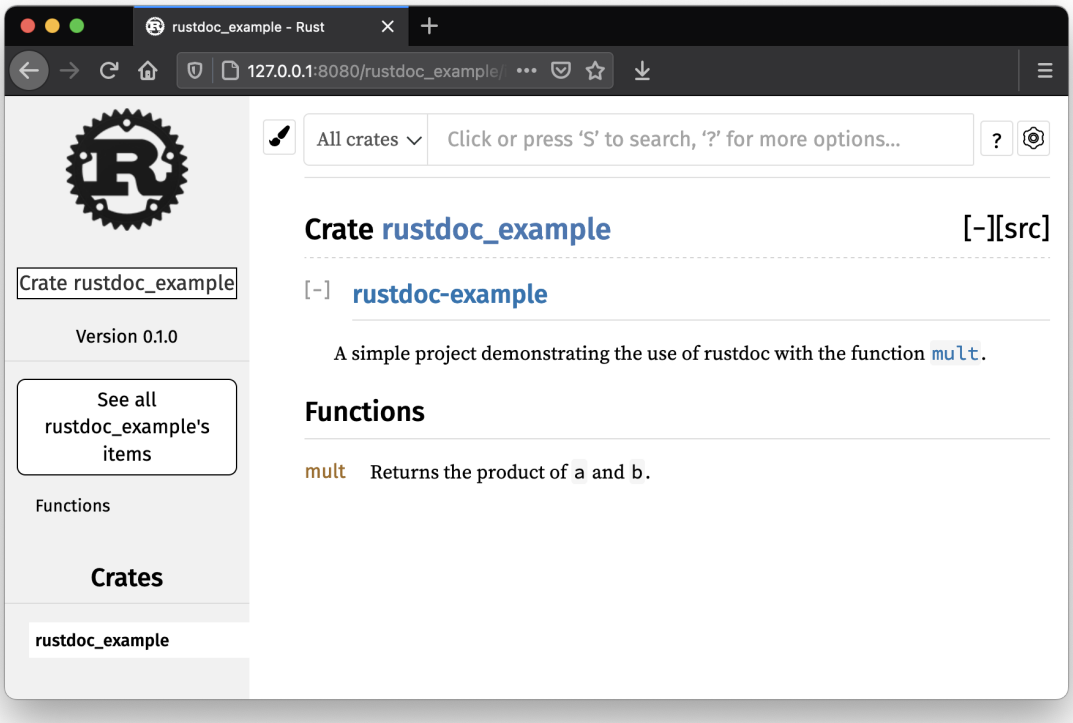

Figure 2.2 Screenshot of `rustdoc` HTML output with comments

For crates published to `crates.io`, there's a companion `rustdoc` site that automatically generates and hosts documentation for crates at https://docs.rs. For example, the docs for the `dryoc` crate are available at https://docs.rs/dryoc.

In documented crates, you should update `Cargo.toml` to include the `documentation` property which links to the documentation for the project. This is helpful for people who find your crate on sources like `crates.io`. For example, the `dryoc` crate has the following in `Cargo.toml`:

```
[package]
name = "dryoc"
documentation = "https://docs.rs/dryoc"
```

You don't have to do anything else to use `docs.rs`, the website automatically generates updated docs when new releases are published to `crates.io`.

Table 2.4 Quick reference for `rustdoc` usage

Syntax	Type	Description
`//!`	Doc string	Crate or module level documentation, belongs at the top of crate or module. Uses commonmark.
`///`	Doc string	Documents the module, function, trait, or type following the comment. Uses commonmark.
`[func]`, `[`func`]`, `[Foo](Bar)`	Link	Links to a function, module, or other type in the docs. The keyword must be in scope for `rustdoc` to link correctly. Many options are available for linking, consult the `rustdoc` documentation for details.

2.8.1 Code examples in documentation

One handy feature of `rustdoc` is that code included as examples within documentation is compiled and executed as integration tests. Thus, you can include code samples with assertions which are tested when you run `cargo test`. This helps you maintain high quality documentation with working code samples.

One such example might look like this (appended to the crate level docs in `src/lib.rs` from the previous example):

```
//! # Example
//!
//! -----
//! use rustdoc_example::mult;
//! assert_eq!(mult(10, 10), 100);
//! -----
```

Running the tests with the example above using `cargo test` yields the following:

```
cargo test
   Compiling rustdoc-example v0.1.0
   (/Users/brenden/dev/code-like-a-pro-in-rust/code/c2/2.8/rustdoc-example)
    Finished test [unoptimized + debuginfo] target(s) in 0.42s
     Running target/debug/deps/rustdoc_example-bec4912aee60500b

running 1 test
test tests::it_works ... ok

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

   Doc-tests rustdoc-example

running 1 test
test src/lib.rs - (line 7) ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out;
  finished in 0.23s
```

For further reference on `rustdoc`, refer to the official documentation at https://doc.rust-lang.org/rustdoc. For more information about commonmark, refer to https://commonmark.org/help.

2.9 Modules

Rust *modules* provide a way to organize code hierarchically, into a set of distinct units, which can optionally be split into separate source files. Rust's modules combine two features into one: the inclusion of code from other source files, and namespacing of publicly visible symbols. In Rust, all symbols are declared as private by default, but they can be exported (or made publicly visible) with the `pub` keyword. If we were to export too many symbols, we might eventually have a name collision. Thus, we can organize our code by modules to prevent namespace pollution.

A module declaration block uses the `mod` keyword, with an optional `pub` visibility specifier, and is followed immediately by a code block in braces:

```
mod private_mod {
  // private code goes here ...
}
pub mod public_mod {
  // public code to be exported goes here
}
```

The terms *module* and *mod* are sometimes used interchangably when talking about Rust code. By convention, module names typically use snake case, whereas most other names use camel case (structs, enums, traits). Primitive types (`i32`, `str`, `u64`, etc) are usually short single words and sometimes snake case. Constants are typically uppercase, which is also the convention in most other languages. Following these patterns makes it easier to determine what's being imported just by glancing at `use` statements.

We can include a module with the same `mod` keyword, except instead of a code block it simply ends with a semicolon:

```
mod private_mod;
pub mod public_mod;
```

Modules can be deeply nested:

```
mod outer_mod {
  mod inner_mod {
    mod super_inner_mod {
      // ...
    }
  }
}
```

SIDEBAR **Visibility specifiers**

In Rust, everything is private by default with respect to visibility, except for public traits and public enums, where associated items are public by default. Privately-scoped declarations are bound to a module, as in they can be accessed from within the module (and submodules) they're declared in.

Using the `pub` keyword changes the visibility to public, with an optional modifier: we can use `pub(`*`modifier`*`)` with `crate`, `self`, `super`, or `in path` with a path to another module. In other words, `pub(crate)` specifies an item is public within the crate, but not accessible outside the crate.

Items declared within a module aren't exported outside the scope of a crate unless the module itself is also public. For example, in the code below we have 2 public functions, but only `public_mod_fn()` in this case would be visible outside the crate:

```
mod private_mod {
  pub fn private_mod_fn() {}
}
pub mod public_mod {
  pub fn public_mod_fn() {}
}
```

Additionally, a privately-scoped item within a public mod is still private and can't be accessed outside its crate.

Rust's visibility is quite intuitive, and also helps to prevent accidentally leaky abstractions. For more detail on Rust's visibilty, refer to the language reference at https://doc.rust-lang.org/reference/visibility-and-privacy.html.

When we include a symbol or module from another crate, we do so with the `use` statement like so:

```
use serde::ser::{Serialize, Serializer};  ❶
```

❶ Includes the `Serialize` and `Serializer` symbols from the 'ser' mod within the `serde` crate.

When we include code with a `use` statement, the first name is usually the name of the crate we want to include code from, followed by a module, specific symbols, or a wildcard (`*`) to include all symbols from that module.

Modules can be organized using the file system, too. We can create the same hierarchy as above using paths within our crate's source directory, but we still need to tell cargo which files to include in the crate. To do this, we use the `mod` *statement*, rather than a block. Consider a crate with the following structure:

```
$ tree .
.
 Cargo.lock
 Cargo.toml
 src
     lib.rs
     outer_module
        inner_module
            mod.rs
            super_inner_module.rs
     outer_module.rs

3 directories, 6 files
```

Above we have a crate with 3 nested inner modules, like in the earlier example. In our top level `lib.rs`, we'll include the outer module which is defined in `outer_module.rs`:

```
mod outer_module;
```

The compiler will look for the mod declaration in either `outer_module.rs` or `outer_module/mod.rs`. In our case, we supplied `outer_module.rs` at the same level as `lib.rs`. Within `outer_module.rs` we have the following to include the inner module:

```
mod inner_module;
```

The compiler next looks for `inner_module.rs` or `inner_module/mod.rs` within the outer module. In this case, it finds `inner_module/mod.rs`, which contains the following:

```
mod super_inner_module;
```

Which includes `super_inner_module.rs` within the `inner_module` directory. This seems quite a bit more complex than example from earlier in this section, but for larger projects it's much better to use modules than including all the source code for a crate in either `lib.rs` or `main.rs`. If modules seem a bit confusing, try recreating similar structures from scratch to understand how the pieces fit together. You can start with the example included in this book's source code under `c02/modules`. We'll also explore module structures again in chapter 9.

2.10 Workspaces

Cargo's workspace feature allows you to break a large crate into separate crates. Workspaces have a few important features which we'll discuss in this section. You will most likely encounter workspaces from other crates, even if you don't use workspaces in your own projects. Projects within a workspace share the following:

- A top-level Cargo.lock file
- The `target/` output directory, containing project targets from all workspaces
- `[patch]`, `[replace]`, and `[profile.*]` sections from the top-level `Cargo.toml`

To use workspaces, you'll create projects with Cargo as you normally would, within

subdirectories that don't overlap with the top level crate's directories (i.e., they shouldn't be in `src/`, `target/`, `tests/`, `examples/`, `benches/`, etc). You can then add these dependencies as you normally would, except rather than specifying a version or repository, you simply specify a path.

Let's walk through an example project using workspaces. Start by creating a top-level application, and change into the newly created directory:

```
$ cargo new workspaces-example
     Created binary (application) `workspaces-example` package
$ cd workspaces-example
```

Now create a subproject, which will be a simple library:

```
$ cargo new subproject --lib
```

The newly created directory structure should look like this:

```
$ tree
.
 Cargo.toml
 src
    main.rs
 subproject
     Cargo.toml
     src
         lib.rs

3 directories, 4 files
```

Next, let's update the top level `Cargo.toml` to include the subproject by adding it as a dependency:

```
[dependencies]
subproject = { path = "./subproject" }
```

In the example above, we're adding the subproject by specifying it as a dependency and using the `path` property on it. Another way to specify all the crates in a workspace is to use `[workspace.members]`, which can contain a list of paths or a glob pattern for the workspace members. For larger projects this may be easier than listing each path explicitly. For this example, the alternative code in `Cargo.toml` would look like this:

```
[workspace]

members = [
    "subproject",
]
```

You can now run `cargo check` to make sure everything compiles without any errors. Currently, our top level project doesn't use the code from the subproject, so let's add a function that returns "Hello, world!" and call that from our application. First, update `subproject/src/lib.rs` to

include our `hello_world` function:

```
pub fn hello_world() -> String {
    String::from("Hello, world!")
}
```

Now update `src/main.rs` in the top level application to call this function:

```
fn main() {
    println!("{}", subproject::hello_world());
}
```

Finally, run our new code:

```
$ cargo run
   Compiling subproject v0.1.0
   (/Users/brenden/dev/code-like-a-pro-in-rust/code/c2/2.9/workspaces-examp
   le/subproject)
   Compiling workspaces-example v0.1.0 (/Users/brenden/dev/
     ➥code-like-a-pro-in-rust/code/c2/2.9/workspaces-example)
    Finished dev [unoptimized + debuginfo] target(s) in 0.85s
     Running `target/debug/workspaces-example`
Hello, world!
```

You can repeat the same steps above with as many subprojects as desired, just by substituting a different name for each occurrence of `subproject` above. The full code for the example above can be found under `c02/workspaces-example`.

YOUR TURN **Cargo also supports virtual manifests, which are top-level crates that do not specify a `[package]` section in `Cargo.toml`, and only contain subprojects. This is useful when you want to publish a collection of packages under one top-level crate.**

Many crates use workspaces to break out projects. An additional feature of workspaces is that each subproject may be published as its own individual crate for others to use.

A couple good examples of projects that make use of the workspaces feature is the `rand` [8] crate and the Rocket[9] crate, the latter of which uses a virtual manifest.

For a complete reference of Cargo workspaces, see https://doc.rust-lang.org/cargo/reference/workspaces.html.

2.11 Custom build scripts

Cargo provides a build time feature that allows one to specify *build time* operations in a Rust script. The script contains a single rust `main` function plus any other code you'd like to include, including build dependencies which are specified in a special `[build-dependencies]` section of `Cargo.toml`. The script communicates with Cargo by printing specially formatted commands to stdout, which Cargo will interpret and act upon.

It's worth noting, that although it's called a "script", it's not a script in the sense of being interpreted code. That is to say, the code is still compiled by `rustc` and executed from a binary.

Common uses for build scripts:

- compiling C or C++ code
- running custom pre-processors on Rust code before compiling it
- generating Rust protobuf code using `protoc-rust` [10]
- generating Rust code from templates
- running platform checks, such as verifying the presence of and finding libraries

Cargo normally re-runs the build script every time you run a build, but this can be modified using `cargo:rerun-if-changed`.

Let's walk through a simple "Hello, world!" example using a tiny C library. First, create a new Rust application and change into the directory:

```
$ cargo new build-script-example
$ cd build-script-example
```

Next, let's make a tiny C library with a function that returns the string "Hello, world!". Create a file called `src/hello_world.c`:

```
const char *hello_world(void) {
    return "Hello, world!";
}
```

Now update `Cargo.toml` to include the `cc` crate as a build dependency, and the `libc` crate for C types:

```
[dependencies]
libc = "0.2"
[build-dependencies]
cc = "1.0"
```

Let's create the actual build script, by creating the file `build.rs` at the top level directory (*not* inside `src/`, where the other source files are):

```
fn main() {
    println!("cargo:rerun-if-changed=src/hello_world.c");  ❶
    cc::Build::new()  ❷
        .file("src/hello_world.c")
        .compile("hello_world");
}
```

❶ Instructs Cargo to only re-run the build script when `src/hello_world.c` is modified.

❷ Compiles the C code into a library using the `cc` crate.

Lastly, let's update `src/main.rs` to call the C function from our tiny library:

```
use libc::c_char;
use std::ffi::CStr;

extern "C" {
    fn hello_world() -> *const c_char;  ❶
}

fn call_hello_world() -> &'static str {  ❷
    unsafe {
        CStr::from_ptr(hello_world())
            .to_str()
            .expect("String conversion failure")
    }
}

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

❶ Defines the external interface of the C library.

❷ A wrapper around the external library that extracts the static C string.

Finally, compile and run the code:

```
$ cargo run
   Compiling cc v1.0.67
   Compiling libc v0.2.91
   Compiling build-script-example v0.1.0 (/Users/brenden/dev/
   code-like-a-pro-in-rust/code/c2/2.10/build-script-example)
    Finished dev [unoptimized + debuginfo] target(s) in 2.26s
     Running `target/debug/build-script-example`
Hello, world!
```

The full code for the example above can be found under `c02/build-script-example`.

2.12 Rust projects in embedded environments

As a systems level programming language, Rust is an excellent candidate for embedded programming. This is especially true in cases where memory allocation is explicit and safety is paramount. In this book I won't explore embedded Rust in depth–that's a subject which warrants its own book entirely–but it's worth mentioning in case you're considering Rust for embedded projects.

Rust's static analysis tooling is especially powerful in embedded domains where it can be more difficult to debug and verify code at runtime. Compile time guarantees can make it easy to verify resources states, pin selections, and safely run concurrent operations with shared state.

If you'd like to experiment with embedded Rust, there is excellent support for Cortex-M device emulation using the popular *QEMU* project [11]. Sample code is available at https://github.com/rust-embedded/cortex-m-quickstart.

At the time of writing, embedded Rust resources for non-ARM architectures are limited, but one notable exception is the Arduino Uno platform. The `ruduino` crate [12] provides reusable components specifically for the Arduino Uno, which is an affordable, low-power, embedded platform that can be acquired for the cost of dinner for two. More information on the Arduino platform can be found at https://www.arduino.cc.

Rust's compiler (`rustc`) is based on the *LLVM* project [13], thus any platform for which LLVM has an appropriate backend is technically supported, although peripherals may not necessarily work. For example, there is early support for RISC-V which is supported by LLVM, but hardware options for RISC-V are limited.

To learn more about embedded Rust, a book is available online at https://docs.rust-embedded.org/book.

2.12.1 Memory allocation

For cases where dynamic memory allocation isn't necessary, you can use the `heapless` crate to provide data structures with fixed sizes and no dynamic allocation. If dynamic memory allocation is desired, it's relatively easy to create your own allocator by implementing the `GlobalAlloc` trait [14].

For some embedded platforms, such as the popular Cortex-M processors, there already exists a heap allocator implementation with the `alloc-cortex-m` crate.

2.13 Summary

- Cargo is the primary tool used for building, managing, and publishing Rust projects.
- In Rust, packages are known as crates, and crates can be published as libraries or applications to the `https://crates.io` registry.
- Cargo is used to install crates from `crates.io`.
- Cargo can be used to automate build, test, and publish steps of a continuous integration and deployment (CI/CD) system.
- The `cargo doc` command will automatically generate documentation for a Rust project using `rustdoc`. Documentation can be formatted using the commonmark format (a specification of markdown).
- As with `crates.io`, https://docs.rs provides free documentation hosting automatically for open source crates published to `crates.io`.
- Rust can generate binaries for distribution that include all dependencies, *excluding* the C library. On Linux systems, you should statically link to musl rather than using the system's C library for maximum portability when distributing pre-compiled binaries.
- Crates can be organized into modules and workspaces, which provides a way to separate code into its parts.

3 Rust tooling

This chapter covers

- Introducing core Rust language tools:
  - `rust-analyzer`
  - Rustfmt
  - Clippy
  - `sccache`
- Integrating Rust tools with VS Code
- Using stable versus nightly toolchains
- Exploring additional non-core tools you may find useful

Mastery of any language depends on mastering its tooling. In this chapter, we'll explore some of the critical tools you need to be effective with Rust.

Rust has a number of tools available to improve productivity and reduce the amount of busywork required to produce high quality software. Rust's compiler, `rustc`, is built upon *LLVM*, so Rust inherits the rich tools included with LLVM, such as LLVM's debugger, *LLDB*. In addition to the tools you expect to find from other languages, Rust includes a number of its own Rust-specific tools, which are discussed in this chapter.

The main tools discussed in this chapter are `rust-analyzer`, rustfmt, Clippy, and `sccache`. These are tools you'll likely use every time you work with Rust. Additionally, I have included instructions for a few other tools, which you may find yourself using occasionally: `cargo-update`, `cargo-expand`, `cargo-fuzz`, `cargo-watch`, and `cargo-tree`.

3.1 Overview of Rust tooling

In chapter 2 we focused on working with Cargo, which is Rust's project management tool. Additionally, there are a number of tools that you may want to use when working with Rust. Unlike Cargo, these tools are optional and can be used at your own discretion. However, I find them to be extremely valuable, and I use them on nearly all of my Rust projects. Projects may require some of these tools, so it's worthwhile to familiarize yourself with them.

The tools discussed in this chapter are normally used through a text editor, or as command line tools. In the table 3.1 I've listed a summary of the core Rust language tools, and in table 3.2 I've summarized a few popular editors and their support for Rust.

Table 3.1 Summary of Rust's core language tools

Name	Description
Cargo	Rust's project management tool for compiling, testing, and managing dependencies, covered in chapter 2
`rust-analyzer`	Provides Rust support for text editors that implement the language server protocol
Rustfmt	Rust's opinionated code style tool, which provides automatic code formatting and checking, and can be integrated into CI/CD systems
Clippy	Rust's code quality tool, which provides a plethora of code quality checks (called lints), and can be integrated into CI/CD systems
`sccache`	General-purpose compiler cache tool to improve compilation speed for large projects

Table 3.2 Summary of Rust editors

Editor	Extension	Summary	References
Emacs	`rust-analyzer`	Rust support via LSP	https://emacs-lsp.github.io/lsp-mode/page/lsp-rust-analyzer/
Emacs	`rust-mode`	Native Emacs extensions for Rust	https://github.com/rust-lang/rust-mode
IntelliJ IDEA	IntelliJ Rust	JetBrains native integration for Rust	https://www.jetbrains.com/rust/
Sublime	`rust-analyzer`	Rust support via LSP	https://github.com/sublimelsp/LSP-rust-analyzer
Sublime	Rust enhanced	Native Sublime package for Rust	https://github.com/rust-lang/rust-enhanced
Vim	`rust-analyzer`	Rust support via LSP	https://rust-analyzer.github.io/manual.html#vimneovim
Vim	rust.vim	Native Vim configuration for Rust	https://github.com/rust-lang/rust.vim
VS Code	`rust-analyzer`	Rust support via LSP	https://rust-analyzer.github.io/manual.html#vs-code

3.2 Using *`rust-analyzer`* for Rust IDE integration

`rust-analyzer` is the most mature and full-featured editor tool for the Rust language. `rust-analyzer` can be integrated with any editor which implements the Language Server Protocol[15] (LSP). Some of the features provided by `rust-analyzer` include:

- Code completions
- Import insertion
- Jumping to definitions
- Renaming symbols
- Documentation generation
- Refactorings
- Magic completions
- Inline compiler errors
- Inlay hints for types and parameters
- Semantic syntax highlighting
- Displaying inline reference documentation

With VS Code, `rust-analyzer` can be installed using the CLI:

```
$ code --install-extension rust-lang.rust-analyzer
```

Once installed, VS Code will look as shown in figure 3.1 when working with Rust code. Note the `Run | Debug` buttons at the top of `fn main()`, which allow you to run or debug code with one click.

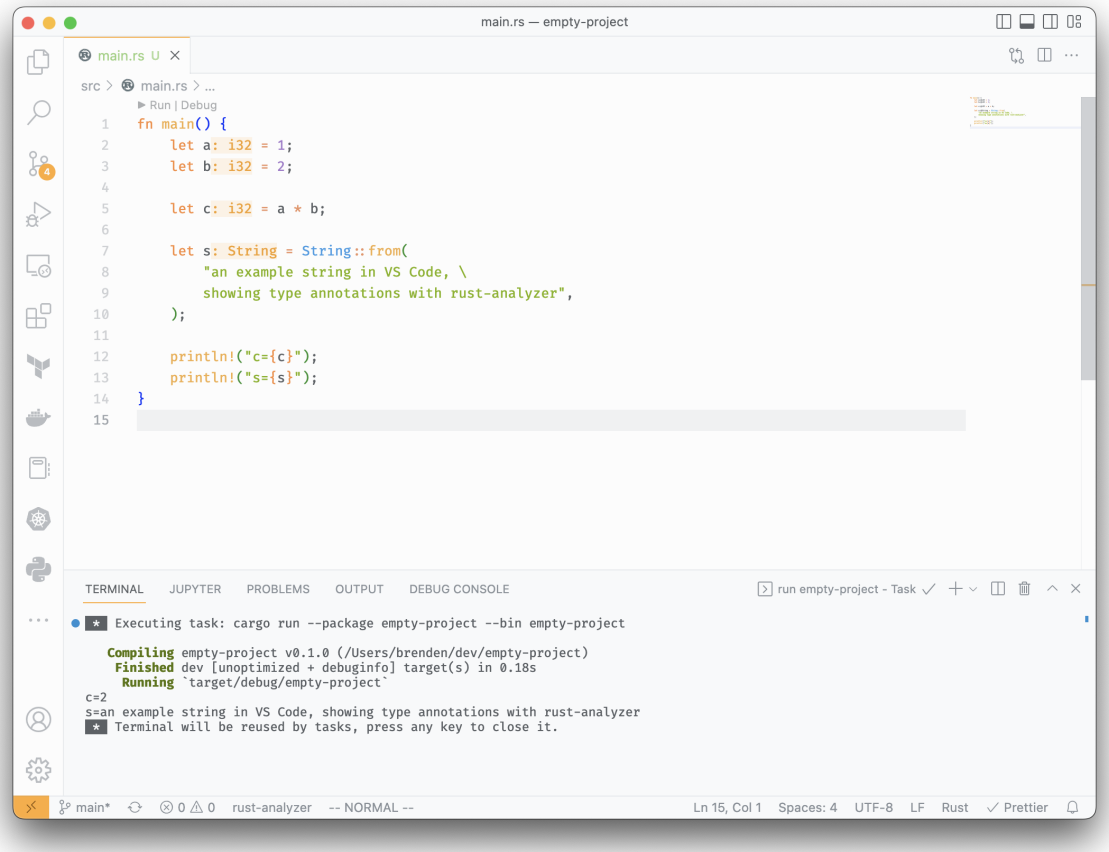

Figure 3.1 VS Code with `rust-analyzer` showing inferred type annotations

If you use IntelliJ Rust, there is no need to install a separate extension for Rust support. However, it's worth noting that IntelliJ Rust shares some code with `rust-analyzer`, specifically for its macro support[16].

3.2.1 Magic completions

`rust-analyzer` has a postfix text completion feature that provides quick completions for common tasks, such as debug printing or string formatting. Becoming familiar with *magic completions* can save you a lot of typing, and reduce some repetitiveness of typing. Additionally, you only need to remember the completion expressions rather than memorizing syntax. I recommend practicing magic completions, as you'll find yourself using them frequently once you get the hang of the syntax.

Magic completions are similar to *snippets* (a feature of VS Code and other editors), but with a few Rust-specific features that make them a bit like snippets++. Magic completions also work in any editor that supports the language server protocol, not just VS Code.

Using magic completions is as simple as typing an expression, and using the editor's completion dropdown menu. For example, to create a test module in the current source file, you can type `tmod` and select the first completion result which will create a test module template like so:

```
tmod ->
#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    fn test_name() {

    }
}
```

The `tmod` completion creates a test module with a single test function, which can be filled out accordingly. In addition to `tmod`, there's a `tfn` completion which creates a test function.

Another useful magic completion is for string printing. Rust versions prior to 1.58.0 did not support string interpolation. To help with the lack of string interpolation, `rust-analyzer` provides several completions for printing, logging, and formatting strings.

> **NOTE** **While string interpolation was added in Rust 1.58.0, this section has been left in the book because it provides a good demonstration of the features of `rust-analyzer`.**

If you type the following into your editor:

```
let bananas = 5.0;
let apes = 2.0;

"bananas={bananas} apes={apes} bananas_per_ape={bananas / apes}"
```

At this point, placing the cursor at the end of the string quote and typing `.print`, will convert the string to the `println` completion option as shown in figure 3.2 below:

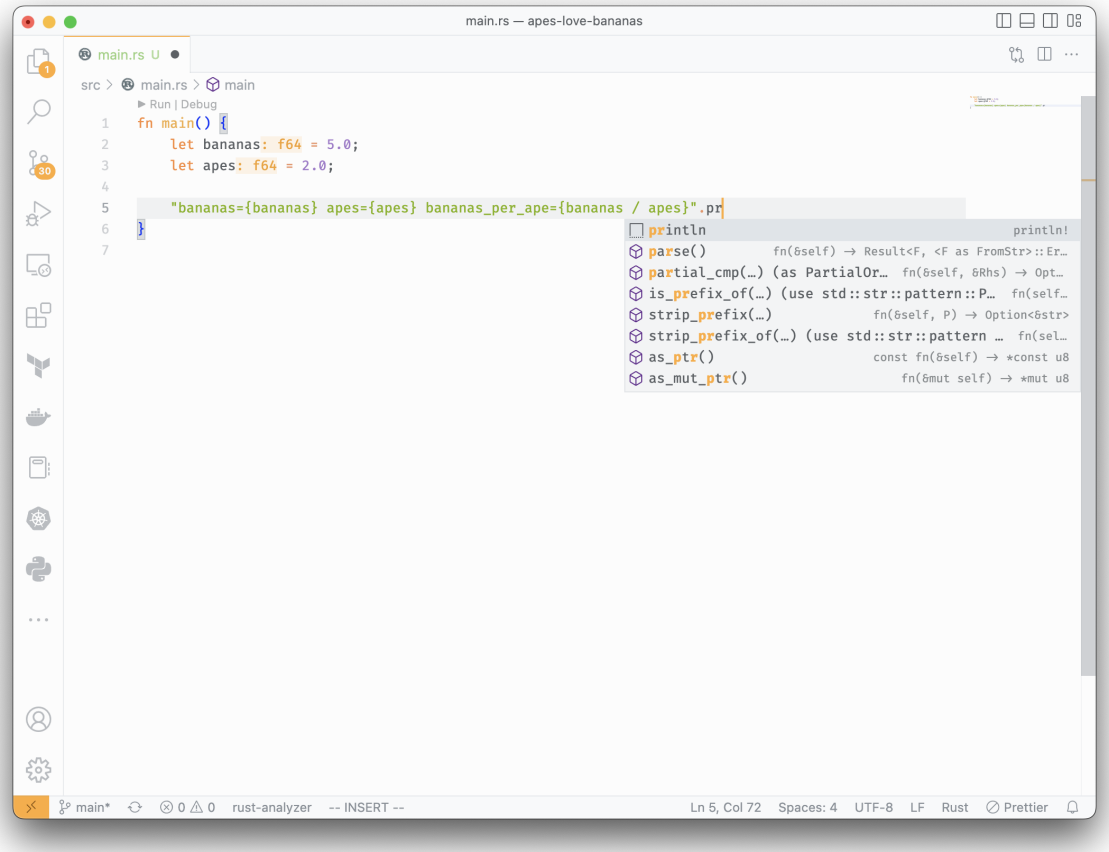

Figure 3.2 VS Code with `rust-analyzer` showing `println` magic completion

If you select the `println` option by pressing the Enter key once the option is selected from the drop down menu that appears, `rust-analyzer` converts the code into the following:

```
let bananas = 5.0;
let apes = 2.0;

println!(
    "bananas={} apes={} bananas_per_ape={}",
    bananas,
    apes,
    bananas / apes
)
```

Table 3.3 Magic completions to remember

Expression	Result	Description
`"str {arg}".format`	`format!("str {}", arg)`	Formats a string with arguments
`"str {arg}".println`	`println!("str {}", arg)`	Prints a string with arguments
`"".logL`	`log::level!("str {}", arg)` where `level` is one of `debug`, `trace`, `info`, `warn`, or `error`	Logs a string with arguments at the specified level
`pd`	`eprintln!("arg = {:?}", arg)`	Debug print (prints to `stderr`) snippet
`ppd`	`eprintln!("arg = {:#?}", arg)`	Debug pretty-print (prints to `stderr`) snippet
`expr.ref`	`&expr`	Borrows `expr`
`expr.refm`	`&mut expr`	Mutably borrows `expr`
`expr.if`	`if expr {}`	Converts an expression to an if statement. Especially useful with `Option` and `Result`

The list above is not exhaustive; the full list of magic completions and other features of `rust-analyzer` are available in the manual at https://rust-analyzer.github.io/manual.html.

3.3 Using rustfmt to keep code tidy

Source code formatting can be a source of frustration, especially when working with other people. For single contributor projects it's not such a big deal, but once you get past 1 contributor there can be a divergence in coding style. *Rustfmt* is Rust's answer to coding style: provide an idiomatic, automatic, and opinionated styling tool. It's similar in nature to gofmt if you're coming from Golang, or the equivalent formatting tool of other languages. The idea of opinionated formatting is relatively new, and–in my humble opinion–a wonderful addition to modern programming languages.

Example output from running `cargo fmt—--check -v` is shown in figure 3.3, which enables verbose mode and check mode. Passing `--check` will cause the command to return non-zero if the formatting is not as expected, which is useful for checking the code format on continuous integration systems.

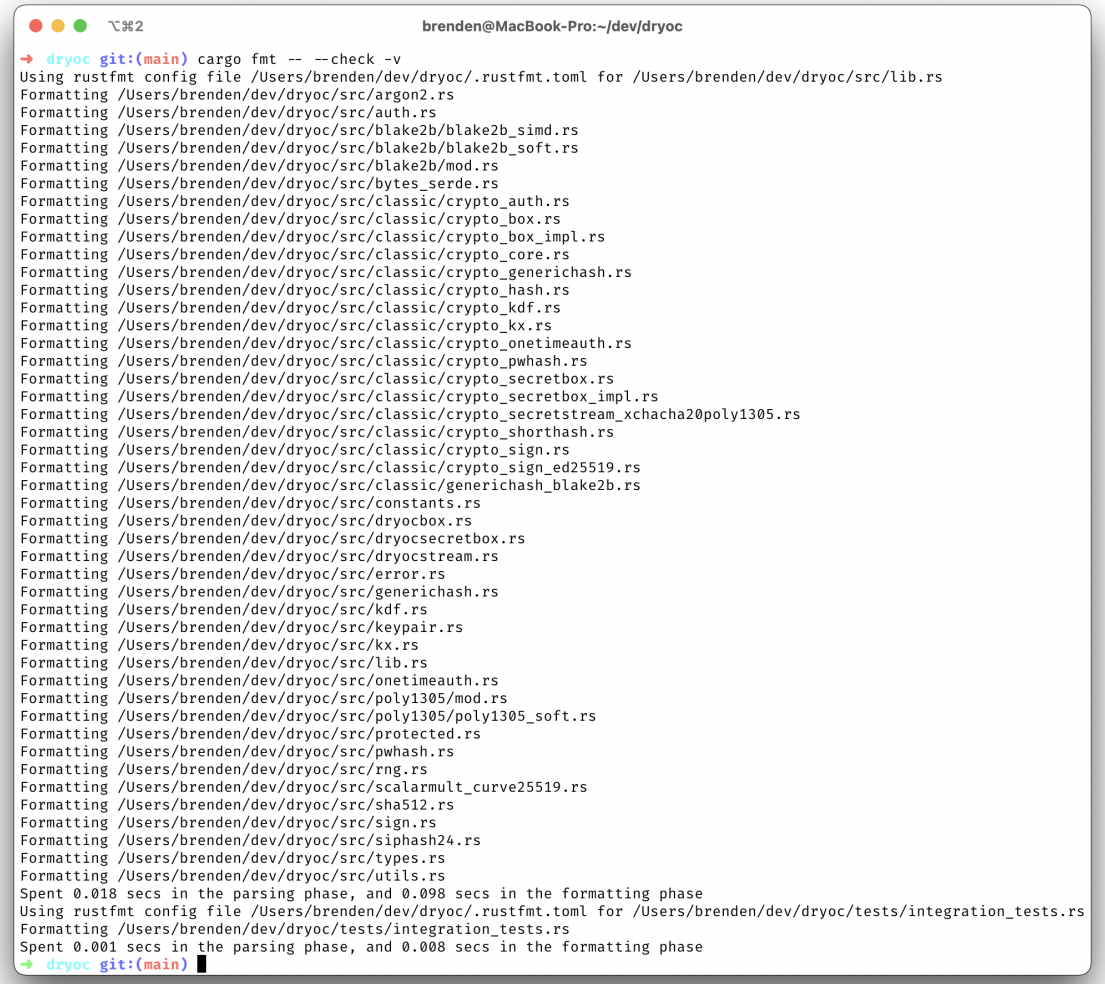

Figure 3.3 Rustfmt in action on the `dryoc` crate

I can't count the number of hours of my life I've lost debating code formatting on pull requests. This problem can be solved instantly by using rustfmt, and simply mandating code contributions follow the defined style. Rather than publishing and maintaining lengthy style guideline documents, you can use rustfmt and save everyone a lot of time.

3.3.1 Installing rustfmt

Rustfmt is installed as a `rustup` component:

```
$ rustup component add rustfmt
...
```

Once installed, it can be used by running Cargo:

```
$ cargo fmt
# Rustfmt will now have formatted your code in-place
```

3.3.2 Configuring rustfmt

While the default rustfmt configuration is going to be fine for the majority of people, you *may* want to tweak settings slightly to suit your preferences. This can be done by adding a `.rustfmt.toml` configuration file to your project's source tree.

Listing 3.1 Example `.rustfmt.toml` configuration

```
format_code_in_doc_comments = true
group_imports = "StdExternalCrate"
imports_granularity = "Module"
unstable_features = true
version = "Two"
wrap_comments = true
```

Table 3.4 Partial listing of rustfmt options

Setting	Default	Recommendation	Description
`imports_granularity`	`Preserve`	`Module`	Defines granularity of import statements.
`group_imports`	`Preserve`	`StdExternalGroup`	Defines the ordering of import grouping.
`unstable_features`	`false`	`true`	Enables nightly-only features, unavailable on the stable channel.
`wrap_comments`	`false`	`true`	Automatically word-wraps comments, in addition to code.
`format_code_in_doc_comments`	`false`	`true`	Applies rustfmt to source code samples in documentation.
`version`	`One`	`Two`	Selects the rustfmt version to use. Some rustfmt features are only available in version 2.

Some of the rustfmt options worth mentioning are nightly-only features at the time of writing. An update-to-date listing of the available style options can be found on the rustfmt website at https://rust-lang.github.io/rustfmt/.

YOUR TURN If you're coming from the C or C++ world and want to apply the same opinionated formatting pattern there, be sure to check out the `clang-format` tool as part of LLVM.

3.4 Using Clippy to improve code quality

Clippy is Rust's code quality tool, which provides more than 450 checks at the time of writing. If you've ever been frustrated by a colleague who likes to chime in on your code reviews and point out minor syntax, formatting, and other stylistic improvements, then Clippy is for you. Clippy can do the same job of your colleague, but without any snark, and Clippy will even give you the code change in many cases.

Clippy can, in many cases, find real problems in your code. However, the real benefit of Clippy is that it obviates the need for arguing over code style issues, because it enforces idiomatic style

and patterns for Rust. Clippy is related to, but a little more advanced than rustfmt, discussed in the previous section.

3.4.1 Installing Clippy

Clippy is distributed as a `rustup` component, thus it's installed as such:

```
$ rustup component add clippy
...
```

Once installed, you can run Clippy on any Rust project using Cargo:

```
$ cargo clippy
...
```

When run, Clippy will produce output that looks similar to the `rustc` compiler output, as shown in figure 3.4 below:

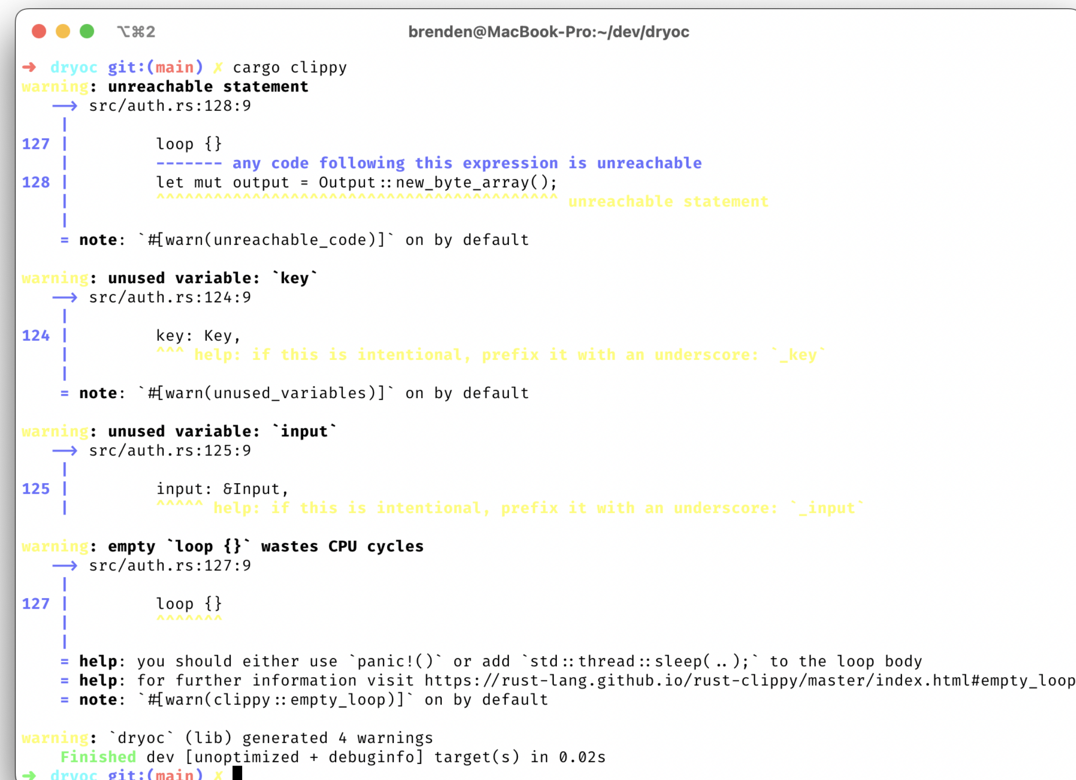

Figure 3.4 Clippy in action on the `dryoc` crate, with an intentional error added

3.4.2 Clippy's lints

With more than 450 code quality checks (known as *lints*), one could write an entire book about Clippy. Lints are categorized by their severity level (allow, warn, deny, deprecated) and grouped according to their type, which can be one of: correctness, restriction, style, deprecated, pedantic, complexity, perf, cargo, and nursery.

One such lint is the `blacklisted_name` lint, which disallows the use of variable names such as `foo`, `bar`, or `quux`. The lint can be configured to include a custom list of variable names you wish to disallow.

Another example of a lint is the `bool_comparison` lint, which checks for unnecessary comparisons between expressions and booleans. For example, the following code is considered invalid:

```
if function_returning_boolean() == true {}  (1)
```

(1) Clippy will complain here.

Whereas, the following code is valid:

```
if function_returning_boolean() {}  (1)
```

(1) The `== true` is not necessary.

The majority of Clippy's lints are style related, but it can also help find performance issues. For example, the `redundant_clone` lint can find situations where a variable is cloned without need. Typically this case looks something like this:

```
let my_string = String::new("my string");
println!("my_string='{}'", my_string.clone());
```

In the code above, the call to `clone()` is entirely unnecessary. If you run Clippy with this code, you'll get the following warning:

```
$ cargo clippy
warning: redundant clone
 --> src/main.rs:3:37
  |
3 | println!("my_string='{}'", my_string.clone());
  |                                     ^^^^^^^^ help: remove this
  |
  = note: `#[warn(clippy::redundant_clone)]` on by default
note: this value is dropped without further use
 --> src/main.rs:3:28
  |
3 | println!("my_string='{}'", my_string.clone());
  |                            ^^^^^^^^^
  = help: for further information visit
  https://rust-lang.github.io/rust-clippy/master/index.html#redundant_clone

warning: 1 warning emitted
```

Clippy is frequently updated, and an up-to-date list of the lints for stable Rust can be found in the Clippy documentation at https://rust-lang.github.io/rust-clippy/stable/index.html.

3.4.3 Configuring Clippy

Clippy can be configured either by adding a `.clippy.toml` file to the project source tree, or by placing attributes within your Rust source files. In most cases, you'll want to use attributes to disable Clippy lints on an as-needed basis. There are many cases where Clippy may generate a warning, but the code is as intended.

Notably, some complexity warnings from Clippy may need to be tweaked or disabled when there's no better alternative. For example, the `too_many_arguments` warning will trigger when you have a function with more than the default limit of 7 arguments. You could increase the default value, or simply disable it for the specific function:

```
#[allow(clippy::too_many_arguments)]
fn function(
  a: i32, b: i32, c: i32, d: i32, e: i32, f: i32, g: i32, h: i32
) {
  // code goes here
}
```

The `allow()` attribute above is specific to Clippy, and instructs it to allow an exception for the `too_many_arguments` lint on the next line of code.

Alternatively, to change the argument threshold for the entire project, you could add the following into your `.clippy.toml`:

```
too-many-arguments-threshold = 10  ❶
```

❶ Sets the argument threshold to 10, up from the default of 7.

The `.clippy.toml` file is a normal TOML file, that should contain a list of `name = value` pairs, according to your preferences. Each lint and its corresponding configuration parameters

are described in detail in the Clippy documentation at https://rust-lang.github.io/rust-clippy/stable/index.html.

3.4.4 Automatically applying Clippy's suggestions

Clippy can, in some cases, automatically fix code. In particular, when Clippy is able to provide a precise suggestion for you to fix the code, it can generally apply the fix automatically as well. To fix the code automatically, run Clippy with the `--fix` flag:

```
$ cargo clippy --fix -Z unstable-options
...
```

Note that we pass the `-Z unstable-options` option as well, because at the time of writing, the `--fix` feature is nightly-only.

3.4.5 Using Clippy in CI/CD

I recommend enabling Clippy as part of your CI/CD system, provided you have one. You would typically run Clippy as a step after build, test, and format. Additionally, you may want to instruct Clippy to fail on warnings, run for all features, and also check tests.

```
# This command runs Clippy with the default settings
$ cargo clippy
...
# This command runs Clippy, but instructs it to fail on warnings (rather
than allowing warnings)
$ cargo clippy -- -D warnings
...
# This command runs Clippy, fails on warnings, enables all crate features,
and also checks tests (by default Clippy ignores tests)
$ cargo clippy --all-targets --all-features -- -D warnings
...
```

If you maintain an open source project, enabling Clippy as part of the CI/CD checks will make it easier for others to contribute good quality code to your project, and it also makes it easier to confidently maintain the code and accept code changes.

Listing 3.2 Minimal example using Clippy with GitHub Actions

```
on: [push]

name: CI

jobs:
  clippy:
    name: Rust project
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v2
      - name: Install Rust toolchain with Clippy
        uses: actions-rs/toolchain@v1
        with:
          toolchain: stable
          components: clippy
      - name: Run Clippy
        uses: actions-rs/cargo@v1
        with:
          command: clippy
          args: --all-targets --all-features -- -D warnings
```

2 contains a full example of using Clippy and rustfmt together with GitHub's Actions CI/CD system.

3.5 Reducing compile times with *sccache*

The *sccache* tool is a general purpose compiler cache, which can be used with Rust projects. Rust compile times can grow significantly for large projects, and `sccache` helps reduce compile times by caching unchanged objects produced by the compiler. The `sccache` project was created by the nonprofit Mozilla specifically to help with Rust compilation, but it's generic enough to be used with most compilers. It was inspired by the `ccache` tool, which you may have encountered from the C/C++ world.

Even if your project is not large, installing `sccache` and using it locally can save you a lot of time spent recompiling code. To illustrate, compiling the *dryoc* crate from a clean project takes 8.891 seconds on my computer normally. Compiling from a clean project with `sccache` enabled, it takes *5.839* seconds. That's 52% more time to compile a relatively small project with `sccache` versus without. That time accumulates, and can become significant for larger projects.

Note that, `sccache` only helps in cases where code has been previously compiled. It will not speed up fresh builds.

3.5.1 Installing *sccache*

`sccache` is written in Rust, and can be installed using Cargo:

```
$ cargo install sccache
```

Once installed, `sccache` is enabled by using it as a `rustc` wrapper with Cargo. Cargo accepts the

RUSTC_WRAPPER argument as an environment variable. You can compile and build any Rust project using sccache by exporting the wrapper environment variable as such:

```
$ export RUSTC_WRAPPER=`which sccache`  ❶
$ cargo build
...
```

❶ The `which sccache` command returns the path of sccache, assuming it's available in $PATH.

3.5.2 Configuring *sccache*

If you've previously used ccache, then sccache will be familiar. sccache has some noteworthy features which ccache lacks: it can be used directly with a number of networked storage backends, which makes it ideally suited for use with CI/CD systems. It supports the vendor-neutral S3 protocol, a couple of vendor storage services, as well as the open source Redis, and Memcached protocols.

To configure sccache, you can specify environment variables, but it can also be configured through platform-dependant configuration files. By default, sccache uses up to 10GiB of local storage. To configure sccache to use the Redis backend, you can set the address for Redis as an environment variable:

```
# Assuming a redis instance running on the default port at 10.10.10.10,
with a database named `sccache`:
$ export SCCACHE_REDIS=redis://10.10.10.10/sccache
```

For details on sccache configuration and usage, consult the official project documentation at https://github.com/mozilla/sccache.

3.6 Integration with IDEs, including VS Code

This is a topic I won't cover in detail, but it's worth mentioning some features for working with Rust in text editors. These days I prefer to use Visual Studio Code, but it is possible to use tools like rust-analyzer, Clippy, rustfmt, and more with any editor.

For rust-analyzer, there are installation instructions provided in the rust-analyzer manual to integrate with Vim, Sublime, Eclipse, Emacs, and others. rust-analyzer *should* work with any editor that supports the Language Server API, which includes many popular editors in addition to those not mentioned here.

In the case of Visual Studio Code, using rust-analyzer is as simple as installing the extension. From the command line, you need to first make sure you have the rust-src component installed, which you can do with rustup:

```
$ rustup component add rust-src
```

Next, install the VS Code extension from the command line:

```
$ code --install-extension matklad.rust-analyzer
```

Using the extension (in VS Code) is as simple as opening any Rust project in VS Code. It will automatically recognize the `Cargo.toml` file in your project directory, and load the project.

> **YOUR TURN** **You can open VS Code directly from any project directory using the command line by running `code .`, where `.` is just shorthand for the current working directory.**

3.7 Using toolchains: stable vs nightly

You may start out using Rust on the stable toolchain, and slowly find yourself discovering features you *want* to use, but cannot because they aren't available in stable. Those features, however, *are* available in the nightly channel. This is a common issue in Rust, and something many have balked at. In fact, a number of popular crates are *nightly only* crates. That is, they can *only* be used with nightly.

There is, in a sense, two Rusts: stable Rust and nightly Rust. It may sound confusing, or cumbersome, but in practice it's not so bad. In most cases you should be fine using stable, but in some cases you'll want to use nightly. If you're publishing public crates, you may find that you have certain features behind a `nightly` feature flag, which is a common pattern.

You may eventually find yourself asking: why not just use nightly exclusively to get all the benefits of Rust? Practically speaking, this isn't such a bad idea. The only caveat is the case where you want to publish crates for others to use, and your potential customers are only able to use stable Rust. In that case, it makes sense to try and maintain stable support, with nightly features behind a feature flag.

3.7.1 Nightly-only features

You may find need to use nightly-only features, and in order to do so you must tell `rustc` which features you want to use. For example, to enable the Allocator API, a feature only available in nightly Rust at the time of writing, you need to enable the `allocator_api` like so:

Listing 3.3 Code listing for `lib.rs` from dryoc

```
#![cfg_attr(
    any(feature = "nightly", all(feature = "nightly", doc)),   ❶
    feature(allocator_api, doc_cfg)   ❷
)]
```

❶ `any()` returns `true` if *any* of the predicates are true, and `all()` returns true if *all* predicates are true. The `doc` attribute is set automatically whenever the code is being analyzed with `rustdoc`.

❷ If the predicate evaluates to `true`, the `allocator_api` and `doc_cfg` features are enabled.

In the code sample above, I've enabled 2 nightly-only features: `allocator_api` and `doc_cfg`. One feature provides custom memory allocation in Rust, the other enables the `doc` compiler attribute, which allows one to configure `rustdoc` from within the code.

YOUR TURN **Rust's built-in attributes are documented at https://doc.rust-lang.org/reference/attributes.html. The `any()` and `all()` predicates are specific to `cfg` and `cfg_attr`, which are the conditional compilation attributes, which are documented at https://doc.rust-lang.org/reference/conditional-compilation.html.**

Also note that in listing 3.3 we're using a feature flag, which means we need to build this crate with the `nightly` feature enabled to enable this code. At the moment there isn't a way to detect which channel your code is being compiled with, so we have to specify feature flags instead.

3.7.2 Using nightly on published crates

In the *dryoc* crate, I use this pattern to provide a *protected memory* feature. Protected memory, in the case of the dryoc crate, is a feature whereby data structures that use a custom allocator (which is a nightly-only API in Rust, at the time of writing) in order to implement memory locking and protection. The feature gating within the crate looks like this:

Listing 3.4 Code listing from `src/lib.rs`

```
#[cfg(any(feature = "nightly", all(doc, not(doctest))))]
#[cfg_attr(all(feature = "nightly", doc), doc(cfg(feature = "nightly")))]
#[macro_use]
pub mod protected;
```

There are a few things going on in the code above, which I'll explain. First, you'll notice the `doc` and `doctest` keywords. Those are included because I want to make sure the `protected` module is included when building the documentation, but *not* when running the doctests if `feature = "nightly"` isn't enabled (i.e., testing the code samples within the crate documentation). The first line translates to: enable the next block of code (which is `pub mod protected`) only if `feature = "nightly"` is enabled, or `doc` is enabled and we're *not* running the doctests. `doc` and `doctests` are special attributes that are enabled only while running either `cargo doc`, or `cargo test`.

The second line enables a `rustdoc` specific attribute that tells `rustdoc` to mark all the content

within the module as `feature = "nightly"`. In other words, if you look at the docs for the dryoc crate at https://docs.rs/dryoc/latest/dryoc/protected/index.html you will see a note that says:

> *This is supported on **crate feature `nightly`** only.*

For details on the allocator API feature in Rust, refer to the GitHub tracking issue at https://github.com/rust-lang/rust/issues/32838. For details about the doc attribute feature, refer to the GitHub tracking issue at https://github.com/rust-lang/rust/issues/43781.

3.8 Additional tools: *cargo-update*, *cargo-expand*, *cargo-fuzz*, *cargo-watch*, *cargo-tree*

There are a few more Cargo tools worth mentioning, which I will summarize below. Each of them is supplemental to the tools already discussed, and they may be mentioned elsewhere in the book.

3.8.1 Keeping packages update to date *cargo-update*

Packages installed with Cargo may need to be updated occasionally, and `cargo-update` provides a way to keep them up to date. This is different from project dependencies, which are updated with the `cargo update` command, which is actually part of Cargo.

To install `cargo-update`:

```
$ cargo install cargo-update
```

Using `cargo-update`:

```
# Prints help
$ cargo help install-update
...
# Updates all installed packages
$ cargo install-update -a
...
```

3.8.2 Debugging macros with *cargo-expand*

At some point you may encounter macros in other crates that you need to debug, or you may need to implement your own macro. `rustc` provides a way to generate the resulting source code with the macro applied, and `cargo-expand` is a wrapper around that feature.

To install `cargo-expand`:

```
$ cargo install cargo-expand
```

Using `cargo-expand`, from a project you're working on:

```
# Prints help
$ cargo help expand
...
# Show the expanded form of "outermod::innermod"
$ cargo expand outermod::innermod
...
```

For a simple "Hello, world!" Rust project, the output of `cargo expand` would look like this:

```
#![feature(prelude_import)]
#[prelude_import]
use std::prelude::rust_2018::*;
#[macro_use]
extern crate std;
fn main() {
    {
        ::std::io::_print(::core::fmt::Arguments::new_v1(
            &["Hello, world!\n"],
            &match () {
                () => [],
            },
        ));
    };
}
```

You can run `cargo-expand` for an entire project, or filter by item name as per the example above. It's worth experimenting with `cargo-expand` to see how other code looks once its macros are expanded. For any moderately large project, the expanded code can become very large, so I recommend filtering by specific functions or mods.

I have found `cargo-expand` especially useful when using libraries with macros, when I need to understand what's happening in other people's code.

3.8.3 Testing with *`cargo-fuzz`*

Fuzz testing is one strategy for finding unexpected bugs, and `cargo-fuzz` provides fuzzing support based on LLVM's *libFuzzer* [17].

To install `cargo-fuzz`:

```
$ cargo install cargo-fuzz
```

Using `cargo-fuzz`, from a project you're working on:

```
# Prints help
$ cargo help fuzz
...
```

Using `cargo-fuzz` requires creating tests using the libFuzzer API. This can be accomplished with the `cargo-fuzz` tool by running the `cargo fuzz` *`add`* command, followed by the name of the test. For example, to create a boilerplate test (with `cargo-fuzz`):

```
$ cargo fuzz new myfuzztest   ❶
$ cargo fuzz run myfuzztest   ❷
```

❶ Creates a new fuzz test called "myfuzztest"

❷ Runs the newly created test, which may take a long time

The resulting test (created by `cargo-fuzz`, in `fuzz/fuzz_targets/myfuzztest.rs`) looks like so:

```
#![no_main]
use libfuzzer_sys::fuzz_target;

fuzz_target!(|data: &[u8]| {
    // fuzzed code goes here
});
```

Fuzz testing is covered in more detail in chapter 7. If you're already familiar with libFuzzer or fuzz testing in general, you should have no trouble getting up to speed on your own with `cargo-fuzz`.

3.8.4 Iterating with *`cargo-watch`*

`cargo-watch` is a tool that continuously watches your project's source tree for changes, and executes a command when the there's a change. We previously mentioned `cargo-watch` in chapter 2. Common use cases for `cargo-watch` are automatically running tests, generating documentation with `rustdoc`, or simply recompiling your project.

To install `cargo-watch`:

```
$ cargo install cargo-watch
```

Using `cargo-watch`, from a project you're working on:

```
# Prints help
$ cargo help watch
...
# Runs `cargo check` continuously
$ cargo watch
# Continuously rebuilds documentation on changes
$ cargo watch -x doc
```

3.8.5 Examining dependencies with *`cargo-tree`*

As projects grow in complexity, you may find yourself perplexed by dependencies either because there are too many, or there are conflicts, or some other combination thereof. One tool that's useful for figuring out where dependencies come from is `cargo-tree`.

To install `cargo-tree`:

```
$ cargo install cargo-tree
```

Using `cargo-tree`, from a project you're working on:

```
# Prints help
$ cargo help tree
...
```

As an example, if I run `cargo-tree` on the *dryoc* crate, I will see the dependency tree as shown in listing 3.5.

Listing 3.5 Partial listing of `cargo tree` output for dryoc crate.

```
$ cargo tree
dryoc v0.3.9 (/Users/brenden/dev/dryoc)
 bitflags v1.2.1
 chacha20 v0.6.0
    cipher v0.2.5
       generic-array v0.14.4
           typenum v1.12.0
          [build-dependencies]
           version_check v0.9.2
    rand_core v0.5.1
        getrandom v0.1.16
            cfg-if v1.0.0
            libc v0.2.88
 curve25519-dalek v3.0.2
    byteorder v1.3.4
    digest v0.9.0
       generic-array v0.14.4 (*)
    rand_core v0.5.1 (*)
    subtle v2.4.0
    zeroize v1.2.0
        zeroize_derive v1.0.1 (proc-macro)
            proc-macro2 v1.0.26
               unicode-xid v0.2.1
            quote v1.0.9
               proc-macro2 v1.0.26 (*)
            syn v1.0.68
               proc-macro2 v1.0.26 (*)
               quote v1.0.9 (*)
               unicode-xid v0.2.1
            synstructure v0.12.4
                proc-macro2 v1.0.26 (*)
                quote v1.0.9 (*)
                syn v1.0.68 (*)
                unicode-xid v0.2.1
... snip ...
```

We can see above the hierarchy of regular and dev-only dependencies for the crate. Packages marked with `(*)` are shown with duplicates removed.

3.9 Summary

- Many popular editors include Rust support, either via the Language Server Protocol (LSP) or nature extensions.
- `rust-analyzer` is the canonical Rust language IDE tool, which can be used with any editor that provides support for LSP.
- Using rustfmt and Clippy can boost productivity and improve code quality.
- There are cases where you may want to use nightly-only features in published crates, and when doing so you should place these features behind a feature flag to support stable users.
- `cargo-update` makes it easy to update your Cargo packages.
- `cargo-expand` lets you expand macros to see the resulting code.
- `cargo-fuzz` let's you easily integrate with libFuzzer for fuzz testing.
- `cargo-watch` automates re-running Cargo commands on code changes.
- `cargo-tree` lets you visualize project dependency trees.

4 Data structures

This chapter covers

- Using Rust's core data structures: string, vectors, and maps
- Understanding Rust's types: primitives, structs, enums, and aliases
- Applying Rust's core types effectively
- Converting between data types
- Demonstrating how Rust's primitive types map to external libraries

Up to this point in the book we haven't spent much time talking about the Rust *language* itself. In the previous two chapters we discussed tooling. With that out of the way we can start diving into the Rust language and its features, which we'll focus on for the rest of this book. In this chapter we'll cover the most important part of Rust after its basic syntax: data structures.

When working with Rust, you'll spend a great deal of time interacting with its data structures, as you would any other language. Rust offers most of the features you'd expect from data structures as in any modern programming language, but it does so while offering exceptional safety and performance. Once you get a handle on Rust's core data types you'll find the rest of the language comes into great clarity as the patterns often repeat themselves.

In this chapter we'll discuss how Rust differs from other languages in its approach to data, review the core data types and structures, and discuss how to effectively use them. We'll also discuss how Rust's primitive types map to C types, which allows you to integrate with non-Rust software.

When working with Rust, you'll likely spend most of your time working with 3 core data structures: strings, vectors, and maps. The implementations included with Rust's standard library are fast, full-featured, and will cover the majority of your typical programming use cases. We'll

begin by discussing strings, which are commonly used to represent a plethora of data sources and sinks.

4.1 Demystifying String, str, &str, and &'static str

In my first encounters with Rust, I was a little confused by the string types. So you if you find yourself in a similar position, worry not, for I have good news: while they *seem* complicated, largely due to Rust's concepts of borrowing, lifetimes, and memory management, I can assure it's all very straightforward once you get a handle on the underlying memory layout.

Sometimes you may find yourself with a `str` when you want a `String`, or you end up with `String` but you have a function that wants a `&str`. Getting from one to the other isn't hard, but it may seem confusing at first. We'll discuss all that and more in this section.

It's important to separate the underlying data (a contiguous sequence of characters) from the interface you're using to interact with them. There is only one *kind* of string in Rust, but there are multiple ways to handle a string's allocation and references to that string.

4.1.1 String vs str

Let's start by trying to clarify a few things: first, there *are* indeed 2 separate core string types in Rust (`String` and `str`). And while they *are* technically different types, they are–for most intents and purposes–the same thing. They both represent a UTF-8 sequence of characters of arbitrary length, stored in a contiguous region of memory. The only *practical* difference between `String` and `str` is how the memory is managed. Additionally, in order to understand *all* core Rust types, it's helpful to think about them in terms of how memory is managed. The 2 Rust string types can be thus be summarized as:

- `str`: a stack allocated UTF-8 string, which can be borrowed but cannot be moved or mutated
- `String`: a heap allocated UTF-8 string, which can be borrowed and mutated

In languages like C and C++, the difference between heap and stack allocated data can be blurry, as C pointers don't tell you *how* memory was allocated. At best, they tell you that there's a region of memory of a specific type, which might be valid, and may be anywhere from 0 to N elements in length. In Rust, memory allocation is explicit, thus your *types* themselves usually define *how* memory is allocated, in addition to the number of elements.

In C you can allocate strings on the stack and mutate them, but this is not allowed in Rust without using the `unsafe` keyword. Not surprisingly, this is a major source of programming errors in C.

Let's illustrate some C strings:

```
char *stack_string = "stack string";
char *heap_string  = strndup("heap string");
```

In the code above, we have 2 identical pointer types, pointing to different *kinds* of memory. The first, `stack_string`, is a pointer to stack-allocated memory. Memory allocated on the stack is usually handled by the compiler, and the allocation is essentially instantaneous. `heap_string` is a pointer of the same type, to a *heap* allocated string. `strndup()` is a standard C library function that allocates a region of memory on the heap using `malloc()`, copies the input into that region, and returns the address of the newly allocated region.

> **NOTE** **If we're being pedantic, we might say that `"heap string"` above is initially stack allocated, but converted into a heap allocated string after the call to `strndup()`. You can prove this by examining the binary generated by the compiler, which would contain the literal `heap string` in the binary.**

Now, as far as C is concerned, all strings are the same: they're just contiguous regions of memory of arbitrary length, terminated by a null character (hex byte value `0x00`).

So, if we switch back to thinking about Rust, we can think of `str` as equivalent to the first line above, `stack_string`. `String` is equivalent to the second line, `heap_string`. While this is somewhat of an oversimplification, it's a good model to help us understand strings in Rust.

4.1.2 Using strings effectively

Most of the time, when working in Rust, you're going to be working with either a `String` or `&str`, but never a `str`. The Rust standard library's *immutable* string functions are implemented for the `&str` type, but the *mutable* functions are only implemented for the `String` type.

It's not possible to create a `str` directly, you can only borrow a reference to one. The `&str` type serves as a convenient lowest common denominator, such as when used as a function argument, because you can always borrow a `String` as `&str`.

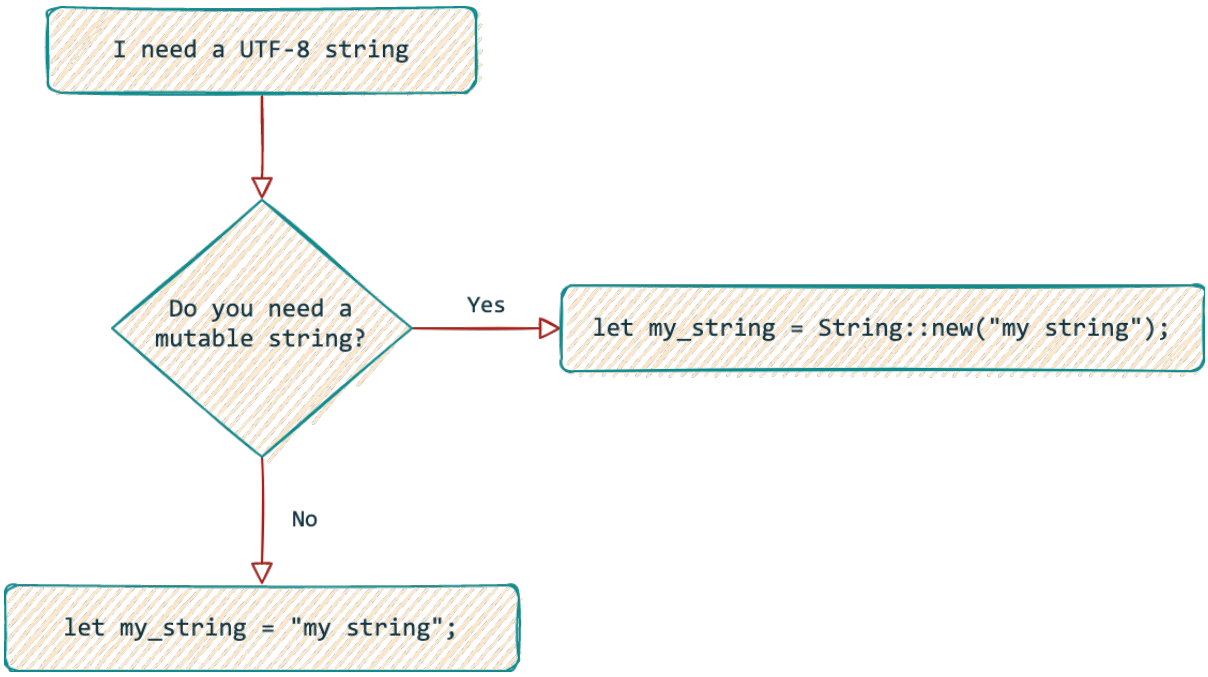

Figure 4.1 Deciding when to use `str` or a `String`, in a very simple flow chart

Let's quickly discuss static lifetimes: In Rust, `'static` is a special lifetime specifier that defines a reference (or borrowed variable) which is valid for the entire life of a process. There are some few special cases where you may need an explicit `&'static str`, but in practice it's something infrequently encountered.

The only real difference between `&'static str` and `&str` is that while a `String` can be borrowed as `&str`, `String` can never be borrowed as `&'static str` because the life of a String is never as long as the process. When a `String` goes out of scope, it's released with the `Drop` trait (we'll explore traits in more detail in chapter 8).

Under the hood, a `String` is actually just a `Vec` of UTF-8 characters. We'll discuss `Vec` in more detail later in the chapter. Additionally, a `str` is just a slice of UTF-8 characters, and we'll discuss slices more in the next section.

Table 4.1 String types summarized

Type	Kind	Components	Use
`str`	Stack-allocated UTF-8 string slice	A pointer to an array of characters, plus its length	Immutable string, such as logging or debug statements, or anywhere else you may have an immutable stack-allocated string.
`String`	Heap-allocated UTF-8 string	A vector of characters	Mutable, resizable string, which can be allocated and deallocated as needed.
`&str`	Immutable string reference	Pointer to either borrowed `str` or `String`, plus its length	Can be used anywhere you want to borrow either a `str` or a `String` immutably.
`&'static str`	Immutable static string reference	Pointer to a `str`, plus its length	A reference to a `str` with an explicit `static` lifetime.

Another difference between `str` and `String` is that `String` can be *moved*, whereas `str` cannot. In fact, it's not possible to own a variable of type `str`, it's only possible to hold reference to a `str`. To illustrate, consider the following example:

Listing 4.1 Movable and non-movable strings

```
fn print_String(s: String) {
    println!("print_String: {}", s);
}

fn print_str(s: &str) {
    println!("print_str: {}", s);
}

fn main() {
    // let s: str = "impossible str";   ❶
    print_String(String::from("String"));   ❷
    print_str(&String::from("String"));   ❸
    print_str("str");   ❹
    // print_String("str");   ❺
}
```

❶ Does not compile, `rustc` will report "error[E0277]: the size for values of type `str` cannot be known at compilation time"

❷ OK: Moves a `String` out of `main` into `print_String`

❸ OK: Returns a `&str` from a `String` in `main`

❹ OK: Creates a `str` on the stack within `main`, and passes a reference to that `str` as `&str` to `print_str`

❺ Does not compile, `rustc` will report "error[E0308]: mismatched types, expected struct `String`, found `&str`"

The code above, when run, prints the following output:

```
print_String: String
print_str: String
print_str: str
```

4.2 Understanding slices and arrays

Slices and *arrays* are special types in Rust. They represent a sequence of arbitrary values of the same type. You can also have multi-dimensional slices or arrays (i.e., slices of slices, or arrays of arrays, or arrays of slices, or slices of arrays).

Slices are a somewhat new programming concept, as you generally won't find the term "slice" used when discussing sequences in the language syntax for Java, C, C++, Python, or Ruby. Typically, sequences are referred to as either arrays (as in Java, C, C++, Ruby), lists (as in Python), or even just a sequence (as in Scala). Other languages may provide equivalent behaviour, but *slices* are not necessarily a first class language concept or type in the way they are in Rust or Go (although the slice abstraction has been catching on in other languages). C++ does have `std::span` and `std::string_view` which provide equivalent behaviour, but the term *slice* is not used in C++ when describing these.

> **NOTE** The term "slices" appears to have originated with the Go language, as described in this blog post from 2013 by Rob Pike: https://go.dev/blog/slices.

In Rust, specifically, slices and arrays differ in a subtle way. An array is a fixed-length sequence of values, and a slice is a sequence of values with an arbitrary length. That is, a slice can be of a variable length, determined at runtime, whereas an array has a fixed-length known at compile time. Slices have another interesting property, which is that you can recursively deconstruct slices into non-overlapping sub-slices.

Working with arrays can at times be tricky in Rust, because knowing the length of a sequence at compile time requires the information be passed to the compiler at compile time, and present in the type signature. As of Rust 1.51, it's possible to use a feature called *const generics* (discussed in more detail in chapter 10) to define generic arrays of arbitrary length, but only at compile time.

Let's illustrate the difference between slices and arrays with some code:

Listing 4.2 Creating an array and a slice

```
let array = [0u8; 64];    ❶
let slice: &[u8] = &array;    ❷
```

❶ The type signature here is [u8; 64], an array, initialized with zeroes.

❷ This borrows a slice of the array.

In the code above, we've initialized a byte array, containing 64 elements, all of which are zero. `0u8` is shorthand for an unsigned integral type, 8 bits in length, with a value of 0. `0` is the value,

and `u8` is the type.

On the second line, we're borrowing the array as a slice. Up to now, this isn't particularly interesting. You can do some slightly more interesting things with slices, such as borrowing twice:

```
let (first_half, second_half) = slice.split_at(32);   ❶
println!(
    "first_half.len()={} second_half.len()={}",
    first_half.len(),
    second_half.len()
);
```

❶ Splits and borrows a slice twice, destructuring it into two separate non-overlapping sub-slices

The code above is calling the `split_at()` function, which is part of Rust's core library and implemented for all slices, arrays, and vectors. `split_at()` *destructures* the slice (which is already borrowed from `array`), and gives us two non-overlapping slices that correspond to the first and second half of the original array.

This concept of *destructuring* is important in Rust, because you may find yourself in situations where you need to borrow a portion of an array or slice. In fact, you can borrow the same slice or array *multiple* times using this pattern, as slices don't overlap. One common use case for this is parsing or decoding text or binary data. For example:

```
let wordlist = "one,two,three,four";
for word in wordlist.split(',') {
    println!("word={}", word);
}
```

Looking at the code above, it may be immediately obvious that we've taken a string, split it on `,` and then we're printing each word within that string. The output from this code prints:

```
word=one
word=two
word=three
word=four
```

What's worth noting about the code above is that there's no heap allocation happening. All of the memory is allocated on the stack, of a fixed length known at compile time, with no calls to `malloc()` under the hood. This is the equivalent of working with raw C pointers, but there's no reference counting or garbage collection involved, thus none of the overhead. And unlike C pointers, the code is succinct, safe, and not overly verbose.

Slices, additionally, have a number of optimizations for working with contiguous regions of memory. One such optimization is the `copy_from_slice()` method, which works on slices. A call to `copy_from_slice()` from the standard library uses the function `memcpy()` function to

copy memory:

Listing 4.3 Snippet of `slice/mod.rs`, from https://doc.rust-lang.org/src/core/slice/mod.rs.html#3071-3097

```
pub fn copy_from_slice(&mut self, src: &[T])
where
    T: Copy,
{
    // ... snipped ...

    // SAFETY: `self` is valid for `self.len()` elements by definition,
    // and `src` was checked to have the same length. The slices cannot
    // overlap because mutable references are exclusive.
    unsafe {
        ptr::copy_nonoverlapping(
          src.as_ptr(),
          self.as_mut_ptr(),
          self.len()
        );
    }
}
```

In the code above–which comes from Rust's core library–`ptr::copy_nonoverlapping()` is just a wrapper around the C library's `memcpy()`. On some platforms, `memcpy()` has additional optimizations beyond what you might be able to accomplish with normal code. Other optimized functions are `fill()` and `fill_with()`, which both use `memset()` to fill memory.

Let's review the core attributes of arrays and slices:

- An array is a fixed-length sequence of values, with the value known at compile time
- Slices are pointers to contiguous regions of memory, including a length, representing an arbitrary-length sequence of values
- Both slices and arrays can be recursively destructured into non-overlapping sub-slices

4.3 Vectors

Vectors, arguably, are Rust's most important data type (the next most important being `String`, which is based on `Vec`). When working with data in Rust, you'll find yourself frequently creating vectors when you need a resizable sequence of values. If you're coming from C++, you've likely heard the term vectors before, and in many ways Rust's vector type is very similar to what you'd find in C++. Vectors serve as a general-purpose container for just about any kind of sequence.

Vectors are one of the ways in Rust to allocate memory on the heap (another being smart pointers, like `Box`; smart pointers are covered in more detail in chapter 5). Vectors have a few internal optimizations to limit excessive allocations, such as allocating memory in blocks. Additionally, in nightly Rust, you can supply a custom allocator (discussed in more detail in chapter 5) to implement your own memory allocation behaviour.

4.3.1 Diving deeper into *Vec*

`Vec` inherits the methods of slices, because we can obtain a slice referenc from a vector. Rust does not have inheritance in the sense of object oriented programming, but rather `Vec` is a special type that is both a `Vec` and a slice at the same time. For example, if you look at the standard library implementation for `as_slice()`:

Listing 4.4 Snippet of `vec/mod.rs`, from https://doc.rust-lang.org/src/alloc/vec/mod.rs.html#376-379

```
pub fn as_slice(&self) -> &[T] {
    self
}
```

The code above is performing a special conversion that (under normal circumstances) wouldn't work. It's taking `self`, which is `Vec<T>` in the code above, and simply returning it as `&[T]`. If you try to compile the same code yourself, it will fail.

How does this work? Rust provides a trait called *`Deref`* (and its mutable companion *`DerefMut`*), which may be used by the compiler to coerce one type into another implicitly. Once implemented for a given type, that type will also automatically implement all the methods of the dereferenced type. In the case of `Vec`, `Deref` and `DerefMut` is implemented as such in the Rust standard library:

Listing 4.5 Snippet of the `Deref` implementation for `Vec`, from https://doc.rust-lang.org/src/alloc/vec/mod.rs.html#376-379

```
impl<T, A: Allocator> ops::Deref for Vec<T, A> {
    type Target = [T];

    fn deref(&self) -> &[T] {
        unsafe { slice::from_raw_parts(self.as_ptr(), self.len) }
    }
}

impl<T, A: Allocator> ops::DerefMut for Vec<T, A> {
    fn deref_mut(&mut self) -> &mut [T] {
        unsafe { slice::from_raw_parts_mut(self.as_mut_ptr(), self.len) }
    }
}
```

In the code above, dereferencing the vector will coerce it into a slice from its raw pointer and length. It should be noted that such an operation is temporary: that is to say, a slice cannot be resized, and the length is provided to the slice at the time of dereferencing.

If, for some reason, you took a slice of a vector, and resized the vector, the slice's size would not change. This would only be possible in unsafe code, however, because the borrow checker will not let you borrow a slice from a vector and change the vector at the same time. To illustrate:

```
let mut vec = vec![1, 2, 3];
let slice = vec.as_slice();   ❶
vec.resize(10, 0);   ❷
println!("{}", slice[0]);   ❸
```

❶ Returns `&[i32]`, because `vec` is borrowed here.

❷ This is a mutable operation.

❸ This fails to compile.

The code above will fail to compile, as the borrow checker returns this error:

```
error[E0502]: cannot borrow `vec` as mutable because it is also borrowed as
immutable
 --> src/main.rs:4:5
  |
3 |     let slice = vec.as_slice();
  |                 --- immutable borrow occurs here
4 |     vec.resize(10, 0);
  |     ^^^^^^^^^^^^^^^^^ mutable borrow occurs here
5 |     println!("{}", slice[0]);
  |                    -------- immutable borrow later used here
```

4.3.2 Wrapping vectors

Some types in Rust merely wrap a `Vec`, such as the `String`. The `String` type is a `Vec<u8>`, and dereferences (using the `Deref` trait mentioned above) into a `str`:

Listing 4.6 Snippet of `string.rs`, from https://doc.rust-lang.org/src/alloc/string.rs.html#278-280

```
pub struct String {
    vec: Vec<u8>,
}
```

Wrapping vectors is a common pattern, as `Vec` is the preferred way to implement a resizable sequence of any type.

4.3.3 Types related to vectors

In 90% of cases you'll want to use a `Vec`. In the other 10%, you'll probably want to use a `HashMap` (discussed in the next section). Container types other than `Vec` or `HashMap` may make sense in certain situations, or cases where you need special optimization, but most likely a `Vec` will be sufficient, and using another type will not provide noticeable performance improvements. A quote comes to mind:

Programmers waste enormous amounts of time thinking about, or worrying about, the speed of noncritical parts of their programs, and these attempts at efficiency actually have a strong negative impact when debugging and maintenance are considered. We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil. Yet we should not pass up our opportunities in that critical 3%.

– Donald Knuth Computing Surveys, Vol 6, No 4, published December 1974

In cases where you are concerned about allocating excessively large regions of contiguous memory, or you are concerned about *where* the memory is located, you can easily get around this problem by simply stuffing a `Box` into a `Vec`, i.e, using `Vec<Box<T>>`.

With that said, there are several other collection types in Rust's standard library, some of which wrap a Vec internally, and you may occasionally need to use them:

- `VecDeque`: double-ended queue that can be resized, based on `Vec`
- `LinkedList`: doubly-linked list
- `HashMap`: a hash map, discussed in more detail in the next section
- `BTreeMap`: a map based on a B-Tree
- `HashSet`: a hash set, based on `HashMap`
- `BTreeSet`: a B-Tree set, based on `BTreeMap`
- `BinaryHeap`: a priority queue, implemented with a binary heap, using a `Vec` internally

Additional recommendations including up-to-date performance details of Rust's core data structures can be found in the Rust standard library collections reference at https://doc.rust-lang.org/std/collections/index.html.

YOUR TURN **It's also reasonable to build your own data structures on top of `Vec`, should you need to. For an example of how to do this, the `BinaryHeap` from Rust's standard library provides a complete example, which is documented at https://doc.rust-lang.org/stable/src/alloc/collections/binary_heap.rs.html.**

4.4 Maps

`HashMap` is the other container type in Rust that you'll find yourself using. If `Vec` is the preferred resizable type of the language, `HashMap` is the preferred type for cases where you need a collection of items that can be retrieved in constant time using a key. Rust's `HashMap` is not much different from hash maps you may have encountered in other languages, but Rust's implementation is probably faster and safer than what you might find in many other libraries.

`HashMap` uses the SipHash 1-3 function for hashing, which is also used in Python (starting from 3.4), Ruby, Swift, and Haskell. This function provides good tradeoffs for common cases, but it

may be inappropriate for very small or very large sized keys, such as integral types or very large strings.

It's also possible to supply your own hash function for use with `HashMap`. You may want to do this in cases where you want to hash very small or very large key, but for most cases the default implementation is adequate.

4.4.1 Custom hashing functions

To use a `HashMap` with a custom hashing function, you need to first find an existing implementation, or write a hash function that implements the necessary traits. `HashMap` requires that `std::hash::BuildHasher`, `std::hash::Hasher`, and `std::default::Default` are implemented for the hash function you wish to use. Traits are discussed in more detail in chapter 8.

Let's examine the implementation of `HashMap` from the standard library:

Listing 4.7 Snippet of `HashMap` from https://doc.rust-lang.org/src/std/collections/hash/map.rs.html#592-596

```
impl<K, V, S> HashMap<K, V, S>
where
    K: Eq + Hash,
    S: BuildHasher,
{
    ❶
}
```

❶ Code intentionally omitted.

Above, you can see `BuildHasher` specified as a trait requirement on the `S` type parameter. Digging a little deeper, you can see `BuildHasher` is just a wrapper around the `Hasher` trait:

Listing 4.8 Snippet of `BuildHasher` from https://doc.rust-lang.org/src/core/hash/mod.rs.html#430-433

```
pub trait BuildHasher {
    /// Type of the hasher that will be created.
    type Hasher: Hasher;   ❶

      ❷
}
```

❶ Here there's a requirement on the `Hasher` trait.

❷ Code intentionally omitted.

The `BuildHasher` and `Hasher` APIs leave most of the implementation details up to the author of the hash function. For `BuildHasher`, only a `build_hasher()` method is required, which returns

the new `Hasher` instance. The `Hasher` trait only requires 2 methods: `write()` and `finish()`. `write()` takes a byte slice (`&[u8]`), and `finish()` returns an unsigned 64-bit integer representing the computed hash. The `Hasher` trait also provides a number of blanket implementations, which you inherit for free if you implement the `Hasher` trait. It's worth examining the documentation for the traits themselves at https://doc.rust-lang.org/std/hash/trait.BuildHasher.html and https://doc.rust-lang.org/std/hash/trait.Hasher.html to get a clearer picture of how they work.

There are numerous crates available on https://crates.io which already implement a wide variety of hash functions. As an example, let's construct a `HashMap` with MetroHash, an alternative to SipHash designed by J. Andrew Rogers, described at https://www.jandrewrogers.com/2015/05/27/metrohash/. The MetroHash crate already includes the necessary implementation of the `std::hash::BuildHasher` and `std::hash::Hasher` traits, which makes this very easy:

Listing 4.9 Code listing for using `HashMap` with MetroHash

```
use metrohash::MetroBuildHasher;
use std::collections::HashMap;

let mut map = HashMap::<String, String, MetroBuildHasher>::default();   ❶
map.insert("hello?".into(), "Hello!".into());   ❷

println!("{:?}", map.get("hello?"));   ❸
```

❶ Creates a new hashmap instance, using MetroHash.

❷ Inserts a key and value pair into the map, using the `Into` trait for conversion from `&str` to `String`.

❸ Retrieves the value from the map, which returns an Option. The `{:?}` argument to the `println!` macro tells it to format this value using the `fmt::Debug` trait.

4.4.2 Creating hashable types

`HashMap` can be used with arbitrary keys and values, but the keys must implement the `std::cmp::Eq` and `std::hash::Hash` traits. Many traits, such as `Eq` and `Hash`, can be *automatically* derived using the `#[derive]` attribute. Consider the following example:

Listing 4.10 Code listing for a compound key type

```
#[derive(Hash, Eq, PartialEq, Debug)]
struct CompoundKey {
    name: String,
    value: i32,
}
```

The code above represents a compound key, composed of a name and value. We're using the `#[derive]` attribute to derive 4 traits: `Hash`, `Eq`, `PartialEq`, and `Debug`. While `HashMap` only

requires `Hash` and `Eq`, we need to also derive `PartialEq` because `Eq` depends on `PartialEq`. I've also derived `Debug`, which provides automatic debug print methods, which is *extremely* convenient for debugging and testing code.

We haven't discussed `#[derive]` much in this book yet, but it's something you'll use frequently in Rust. We'll go into more detail on traits and `#[derive]` in chapters 8 & 9. For now, you should just think of it as an automatic way to generate trait implementations. These trait implementations have the added benefit in that they're composable: so long as they exist for any subset of types, they can also be derived for a superset of types.

We'll also discuss the use of `#[derive]` in the next section on basic types.

4.5 Rust types: primitives, structs, enums, aliases

Being a strongly typed language, Rust provides a number of ways to model data. At the bottom are primitive types, which handle our most basic units of data like numeric values, bytes, and characters. Moving up from there, we have structs and enums which are used to encapsulate other types. Lastly, aliases let us rename and combine ohter types into new types.

To summarize, in Rust there are 4 categories of types:

- ***Primitive* types**, such as strings, arrays, tuples, and integral types
- ***Structs***, a compound type composed of any arbitrary combination of other types, similar to C structs, for example
- ***Enums***, a special type in Rust, which is somewhat similar to `enum` from C, C++, Java, and other languages
- ***Aliases***, which are syntax sugar for creating new type definitions based on existing types

4.5.1 Using primitive types

Primitive types are provided by the Rust language and core library. These are equivalent to the primitives you'd find in any other strongly typed language, with a few exceptions, which we'll review in this section.

Table 4.2 Summary of primitive types in Rust

Class	Kind	Description
Scalar	Integers	Can be either a signed or unsigned integer, anywhere from 8-128 bits in length (bound to a byte, i.e., 8 bits).
Scalar	Sizes	An architecture specific size type, can be signed or unsigned.
Scalar	Floating point	32 or 64 bit floating point numbers.
Compound	Tuples	Fixed-length collection of types or values, which can be destructured.
Sequence	Arrays	Fixed-length sequence of values of a type that can be sliced.

INTEGER TYPES

Integer types can be recognized by their signage designation (either `i` or `u` for signed and unsigned respectively), followed by the number of bits. Sizes begin with `i` or `u`, followed by the word `size`. Floating point types begin with `f`, follow by the number of bits.

Table 4.3 Summary of integer-type identifiers

Length	Signed identifier	Unsigned identifier	C equivalent
8 bits	`i8`	`u8`	`char` **and** `uchar`.
16 bits	`i16`	`u16`	`short` **and** `unsigned short`.
32 bits	`i32`	`u32`	`int` **and** `unsigned int`.
64 bits	`i64`	`u64`	`long`, `long long`, `unsigned long`, **and** `unsigned long long`, **depending on the platform.**
128 bits	`i128`	`u128`	**Extended integers are non-standard C, but provided as** `__int128` **or** `__uint128` **with GCC and clang.**

The type for an integer literal can be specified by appending the type identifier. For example, `0u8` denotes an unsigned 8 bit integer with a value of 0. Integer values can be prefixed with `0b`, `0o`, `0x` or `b` for binary, octal, hexadecimal, and byte literals. Consider this example, which prints each value as a decimal (base 10) integer:

Listing 4.11 Code listing with integer literals

```
let value = 0u8;
println!("value={}, length={}", value, std::mem::size_of_val(&value));
let value = 0b1u16;
println!("value={}, length={}", value, std::mem::size_of_val(&value));
let value = 0o2u32;
println!("value={}, length={}", value, std::mem::size_of_val(&value));
let value = 0x3u64;
println!("value={}, length={}", value, std::mem::size_of_val(&value));
let value = 4u128;
println!("value={}, length={}", value, std::mem::size_of_val(&value));

println!("Binary (base 2)         0b1111_1111={}", 0b1111_1111);
println!("Octal (base 8)          0o1111_1111={}", 0o1111_1111);
println!("Decimal (base 10)         1111_1111={}", 1111_1111);
println!("Hexadecimal (base 16)   0x1111_1111={}", 0x1111_1111);
println!("Byte literal            b'A'={}", b'A');
```

When we run the code above, we get the following output:

Listing 4.12 Output from listing 4.10

```
value=0, length=1
value=1, length=2
value=2, length=4
value=3, length=8
value=4, length=16
Binary (base 2)         0b1111_1111=255
Octal (base 8)          0o1111_1111=2396745
Decimal (base 10)         1111_1111=11111111
Hexadecimal (base 16)   0x1111_1111=286331153
Byte literal            b'A'=65
```

SIZE TYPES

For size types, the identifiers are `usize` and `isize`. These are platform dependent sizes, which are typically 32 or 64 bits in length for 32 and 64 bit systems respectively. `usize` is equivalent to C's `size_t`, and `isize` is provided to permit signed arithmetic with sizes. In the Rust standard library, functions returning or expecting a length parameter expect a `usize`.

ARITHMETIC ON PRIMITIVES

Many languages permit unchecked arithmetic on primitive types. In C and C++, in particular, many arithmetic operations have undefined results and produce no errors. One such example is division by zero. Consider the following C program:

Listing 4.13 Code listing of `divide_by_zero.c`

```
#include <stdio.h>

int main() {
    printf("%d\n", 1 / 0);
}
```

If you compile and run this code with `clang divide_by_zero.c && ./a.out`, it will print a value that appears random. Both clang and gcc happily compile this code, and they both print a warning, but there is no runtime check for an operation which is undefined.

In Rust, all arithmetic is checked by default. Consider the following Rust program:

```
// println!("{}", 1 / 0);  ❶

let one = 1;
let zero = 0;
// println!("{}", one / zero);  ❷

let one = 1;
let zero = one - 1;
// println!("{}", one / zero);  ❸

let one = { || 1 }();
let zero = { || 0 }();
println!("{}", one / zero);  ❹
```

❶ Does not compile

❷ Does not compile

❸ Still doesn't compile

❹ Code panics here!

In the code above, Rust's compiler is pretty good at catching errors at compile time. We have to trick the compiler in order to allow the code to compile and run. Above, we do this by initializing a variable from the return value of a closure. Another way to do it would be to just create a regular function that returns the desired value. In any case, running the problem produces the following output:

```
     Running `target/debug/unchecked-arithmetic`
thread 'main' panicked at 'attempt to divide by zero', src/main.rs:14:20
note: run with `RUST_BACKTRACE=1` environment variable to display a
backtrace
```

If you need more control over arithmetic in Rust, the primitive types provide a number of methods for handling such operations. For example, to safely handle division by zero you can use the `checked_div()` method which returns an `Option`:

```
assert_eq!((100i32).checked_div(1i32), Some(100i32));  ❶
assert_eq!((100i32).checked_div(0i32), None);  ❷
```

❶ 100 / 1 = 1

❷ 100 / 0, result is undefined

For scalar types (integers, sizes, floats) Rust provides a collection of methods that provide basic arithmetic operations (such as division, multiplication, addition, and subtraction) in checked, unchecked, overflowing, and wrapping forms.

When you want to achieve compatibility with the behaviour from languages like C, C++, Java, C# and others, the method you probably want to use is the *wrapping* form, which performs modular arithmetic and is compatible with the C equivalent operations. Keep in mind that overflow on *signed* integers in C is undefined. Here's an example of modular arithmetic in Rust:

```
assert_eq!(0xffu8.wrapping_add(1), 0);
assert_eq!(0xffffffffu32.wrapping_add(1), 0);
assert_eq!(0u32.wrapping_sub(1), 0xffffffff);
assert_eq!(0x80000000u32.wrapping_mul(2), 0);
```

The full listing of arithmetic functions for each primitive is available in the Rust documentation. For `i32`, it can be found at https://doc.rust-lang.org/std/primitive.i32.html.

4.5.2 Using tuples

Rust's *tuples* are similar to what you'll find in other languages. A tuple is a fixed-length sequence of values, and the values can each have different types. Tuples in Rust are not reflective: unlike arrays, you can't iterate over a tuple, take a slice of a tuple, or determine the type of its components at runtime. Tuples are essentially a form of syntax sugar in Rust, and while useful, they are quite limited.

Consider the following example of a tuple:

```
let tuple = (1, 2, 3);
```

The code above looks somewhat similar to what you might expect for an array, except for the limitations mentioned above (can't slice, iterate, or reflect tuples). To access individual elements within the tuple, you can refer to them by their position, starting at `0`:

```
println!("tuple = ({}, {}, {})", tuple.0, tuple.1, tuple.2);  ❶
```

❶ This prints "tuple = (1, 2, 3)"

Alternatively, you can use `match`, which provides temporary destructuring, provided there's a pattern match (pattern matching is discussed in more detail in chapter 8):

```
match tuple {
    (one, two, three) => println!("{}, {}, {}", one, two, three),  ❶
}
```

❶ This prints "1, 2, 3"

We can also destructure a tuple into its parts with the follow syntax, which moves the values *out* of the tuple:

```
let (one, two, three) = tuple;
println!("{}, {}, {}", one, two, three);  ❶
```

❶ This prints "1, 2, 3"

In my experience, the most common use of tuples is returning multiple values from a function. For example, consider this succinct `swap()` function:

```
fn swap<A, B>(a: A, b: B) -> (B, A) {
    (b, a)
}

fn main() {
    let a = 1;
    let b = 2;

    println!("{:?}", swap(a, b));  ❶
}
```

❶ This prints "(2, 1)".

YOUR TURN **It's recommended that you don't make tuples with more than 12 arguments, although there is no strict upper limit to the length of a tuple. The standard library only provides trait implementations for tuples with up to 12 elements.**

4.5.3 Using structs

Structs are the main building block in Rust. They are composite data types, which can contain any set types and values. They are similar in nature to C structs, or classes in object oriented languages. They can be composed generically in a similar fashion to templates in C++ or generics in Java, C#, or TypeScript (generics are covered in more detail in chapter 8).

You should use a struct any time you need to:

- provide stateful functions (i.e., functions or methods that operate on internal-only state)
- control access to internal state (i.e., private variables)
- encapsulate state behind an API

You are not *required* to use structs. You can write APIs with functions only, if you desire, in similar fashion to C APIs. Additionally, structs are only needed to define implementations, they are *not* for specifying interfaces. This differs from object oriented languages like C++, Java, and C#.

The simplest form of a struct is an empty struct:

```
struct EmptyStruct {}

// Unit struct, ends with semicolon with no braces
struct AnotherEmptyStruct;
```

Empty structs are something you may encounter occasionally. They are sometimes used to implement more advanced design patterns which we'll revisit in chapter 10. Another form of struct is the *tuple struct*, which looks like this:

```
struct TupleStruct(String);

let tuple_struct = TupleStruct("string value".into());  ❶
println!("{}", tuple_struct.0);  ❷
```

❶ Initializes the struct similarly to a tuple.

❷ The first tuple element can be accessed with `.0`, second with `.1`, third with `.2`, …

A tuple struct is a special form of struct which behaves like a tuple. The main difference between a tuple struct and a regular struct is that in a tuple struct the values have no names, and only types. Notice how a tuple struct has a semicolon, `;`, at the end of the declaration, which is not required for regular structs (except for an empty declaration).

A typical struct has a list of elements with names and types, like this:

```
struct TypicalStruct {
  name: String,
  value: String,
  number: i32,
}
```

Each element within a struct has *module* visibility by default. That means that values within the struct are accessible anywhere within the scope of the current module. Visibility can be set on a per-element basis:

```
pub struct MixedVisibilityStruct {  ❶
  pub name: String,  ❷
  pub(crate) value: String,  ❸
  pub(super) number: i32,  ❹
}
```

❶ A public struct, visible outside the crate.

❷ This element is public, accessible outside of crate.

❸ This element is public anywhere within the crate.

❹ This element is accessible anywhere within the parent scope.

Most of the time, you shouldn't need to make struct elements public. An element within a struct can be accessed and modified by any code within the public scope for that struct element. The default visibility (which is equivalent to `pub(self)`) allows any code within the same module to access and modify the elements within a struct.

Visibility semantics also apply to the structs themselves, just like their member elements. For a struct to be visible outside a crate (i.e., to be consumed from a library) it must be declared with `pub struct MyStruct { … }`. A struct that's not explicitly declared as public won't be accessible outside the crate (this also applies generally to functions, traits, and any other declarations).

When you declare a struct, you'll probably want to derive a few standard trait implementations:

```
#[derive(Debug, Clone, Default)]
struct DebuggableStruct {
    string: String,
    number: i32,
}
```

Above, we're deriving the `Debug`, `Clone`, and `Default` traits. These traits are summarized as:

- *Debug*: provides a `fmt()` method which formats (for printing) the content of the type
- *Clone*: provides a `clone()` method which creates a copy (or clone) of the type
- *Default*: provides an implementation of `default()`, which returns a default (usually empty) instance of the type

You can derive these traits yourself if you wish (such as in cases where you want to customize their behaviour), but so long as all elements within a struct implement each trait, you can derive them automatically and save a lot of typing.

With these 3 traits derived for the example above, we can now do the following:

```
let debuggable_struct = DebuggableStruct::default();
println!("{:?}", debuggable_struct);   ❶
println!("{:?}", debuggable_struct.clone());   ❷
```

❶ Prints `DebuggableStruct { string: "", number: 0 }`.

❷ Also prints `DebuggableStruct { string: "", number: 0 }`.

To define methods for a struct, you will *implement* them using the `impl` keyword:

```
impl DebuggableStruct {
  fn increment_number(&mut self) {   ❶
    self.number += 1;
  }
}
```

❶ A function that takes a mutable reference to `self`.

The code above takes a mutable reference of our struct and increments it by 1. Another way to do this would be to *consume* the struct, and return it from the function:

```
impl DebuggableStruct {
  fn incremented_number(mut self) -> Self {   ❶
    self.number += 1;
    self
  }
}
```

❶ A function that takes an owned mutable instance of `self`.

There's a subtle difference between these two implementations, but they are functionally

equivalent. There may be cases where you want to consume the input to a method in order to swallow it, but in most cases the first version (using `&mut self`) is preferred.

4.5.4 Using enums

Enums can be thought of as a specialized type of struct which contain enumerated mutually exclusive *variants*. An enum can be *one* of its variants at a given time. With a struct, *all* elements of the struct are present. With an enum, only *one* of the variants is present. An enum can contain any kind of type, not just integral types. The types may be named, or anonymous.

This is quite different from enums in languages like C, C++, Java, or C#. In those languages, enums are effectively used as a way to define constant values. Rust's enums can emulate enums like you might expect from other languages, but they are conceptually different. While C++ has enums, Rust's enums are more similar to `std::variant` than C++'s `enum`.

Consider the following enum:

```
#[derive(Debug)]
enum JapaneseDogBreeds {
    AkitaKen,
    HokkaidoInu,
    KaiKen,
    KishuInu,
    ShibaInu,
    ShikokuKen,
}
```

For the enum above, `JapaneseDogBreeds` is the name of the enum type, and each of the elements within the enum is a unit-like type. Since the types in the enum don't exist outside the enum, they are created *within* the enum. We can run the following code now:

```
println!("{:?}", JapaneseDogBreeds::ShibaInu);   ❶
println!("{:?}", JapaneseDogBreeds::ShibaInu as u32);   ❷
```

❶ This prints "ShibaInu".

❷ This prints "4", the 32-bit unsigned integer representation of the enum value.

Casting the enum type to an u32 works because enum types are enumerated. Now what if we want to go from the number 4 to the enum value? For that, there is no automatic conversion, but we can implement it ourselves using the `From` trait:

```
impl From<u32> for JapaneseDogBreeds {
    fn from(other: u32) -> Self {
        match other {
            other if JapaneseDogBreeds::AkitaKen as u32 == other => {
                JapaneseDogBreeds::AkitaKen
            }
            other if JapaneseDogBreeds::HokkaidoInu as u32 == other => {
                JapaneseDogBreeds::HokkaidoInu
            }
            other if JapaneseDogBreeds::KaiKen as u32 == other => {
                JapaneseDogBreeds::KaiKen
            }
            other if JapaneseDogBreeds::KishuInu as u32 == other => {
                JapaneseDogBreeds::KishuInu
            }
            other if JapaneseDogBreeds::ShibaInu as u32 == other => {
                JapaneseDogBreeds::ShibaInu
            }
            other if JapaneseDogBreeds::ShikokuKen as u32 == other => {
                JapaneseDogBreeds::ShikokuKen
            }
            _ => panic!("Unknown breed!"),
        }
    }
}
```

In the code above, we have to cast the enum type to a `u32` to perform the comparison, and then we return the enum type if there's a match. In the case where no value matches, we call `panic!()` which causes the program to crash. The syntax above uses the match guard feature, which lets us match using an `if` statement.

It's possible to specify the enumeration variant types in an enum, as well. This can be used to achieve behaviour similar to C enums:

```
enum Numbers {
    One = 1,
    Two = 2,
    Three = 3,
}

fn main() {
  println!("one={}", Numbers::One as u32);   ❶
}
```

❶ This prints "1". Note that without the `as` cast, this does not compile because `One` doesn't implement `std::fmt`.

Enums may contain tuples, structs, and anonymous (i.e., unnamed) types as variants:

```
enum EnumTypes {
    NamedType,   ❶
    String,   ❷
    NamedString(String),   ❸
    StructLike { name: String },   ❹
    TupleLike(String, i32),   ❺
}
```

❶ A named type.

❷ An unnamed String type.

❸ A named String type, specified as a tuple with one item.

❹ A struct-like type, with a single element called 'name'.

❺ A tuple-like type with 2 elements.

To clarify, an *unnamed* enum variant is a variant that's specified as a *type*, rather than with a name. A *named* enum variant is equivalent to creating a new type within the enum, which also happens to correspond to an enumerated integer value. In other words, if you want to emulate the behaviour of enums from languages like C, C++, or Java, you'll be using named variants, which conveniently emulate the enumeration behaviour by casting the value to an integer type *even though* enum variants are also types (i.e., not just values).

As a general rule, it's good practice to avoid mixing named and unnamed variants within an enum, as it can be confusing.

4.5.5 Using aliases

Aliases are a special type in Rust that allows you to provide an alias for any other type. They are equivalent to C and C++'s `typedef` or the C++ `using` keyword. Defining an alias does not create a new type.

Aliases have 2 common uses:

- Providing aliased type definitions for public types, as a matter of ergonomics and convenience for the user of a library.
- Providing shorthand types that correspond to more complicated type compositions.

For example, I may want to create a type alias for a hash map which I frequently use within my crate:

```
pub(crate) type MyMap = std::collections::HashMap<String, MyStruct>;
```

Now rather than having to type the full `std::collections::HashMap<String, MyStruct>`, I can just use `MyMap` instead.

For libraries, it's common practice to export public type aliases with sensible defaults for type construction when generics are used. It can be difficult at times to determine which types are required for a given interface, and aliases provide one way for library authors to signal that information.

In the *dryoc* crate, I provide a number of type aliases for convenience. The API makes heavy use of generics. One such example looks like this:

Listing 4.14 Snippet for `kdf.rs` from https://docs.rs/dryoc/0.3.8/src/dryoc/kdf.rs.html#42-45

```
/// Stack-allocated key type alias for key derivation with [`Kdf`].
pub type Key = StackByteArray<CRYPTO_KDF_KEYBYTES>;
/// Stack-allocated context type alias for key derivation with [`Kdf`].
pub type Context = StackByteArray<CRYPTO_KDF_CONTEXTBYTES>;
```

In the code above, the `Key` and `Context` type aliases are provided within this module so the user of this library does not need to worry about implementation details.

4.6 Error handling with *`Result`*

Rust provides a few features to make error handling easier. These features are based on an enum called `Result`, which has the following definition:

Listing 4.15 Snippet of `std::result::Result`, from https://doc.rust-lang.org/std/result/enum.Result.html

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

A `Result` represents an operation which can either succeed (returning a result), or fail (returning an error). You will quickly become accustomed to seeing `Result` as the return type for many functions in Rust.

You will likely want to create your own error type in your crate. That type could either be an enum containing all the different kinds of errors you expect, or simply a struct with something actionable such as an error message. I, being a simple person, prefer to just provide a helpful message and move on with my life. Here's a very simple error struct:

```
#[derive(Debug)]
struct Error {
    message: String,
}
```

Within your crate, you'll need to decide what type of errors you want *your* functions to return. My suggestion is to have your crate return its own error type. This is convenient for anyone else using your crate, because it will be clear to them where the error originates from.

To make this pattern work, you'll need to implement the `From` trait (discussed in the next section) for your error type for any *other* types of errors you may need to convert from. Doing this is relatively easy, because the compiler will tell you when it's necessary.

Now, within your crate, suppose you have a function that reads the contents of a file, like this:

```
fn read_file(name: &str) -> Result<String, Error> {
    use std::fs::File;
    use std::io::prelude::*;

    let mut file = File::open(name)?;   ❶
    let mut contents = String::new();
    file.read_to_string(&mut contents)?;   ❷
    Ok(contents)
}
```

❶ Using the `?` operator here, for implicit error handling.

❷ Using the `?` operator here too.

Above we have a function that opens a file, `name`, reads the contents into a string, and returns the contents as a result. We use the `?` operator twice, which works by returning the result of the function upon success, or returning the error immediately. Both `File::open` and `read_to_string()` use the `std::io::Error` type, and so we've provided the following `From` implementation that permits this conversion automatically:

```
impl From<std::io::Error> for Error {
    fn from(other: std::io::Error) -> Self {
        Self {
            message: other.to_string(),
        }
    }
}
```

4.7 Converting types with From/Into

Rust provides 2 very useful traits as part of its core library: the *`From`* and *`Into`* traits. If you browse the Rust standard library, you may notice that `From` and `Into` are implemented for a great number of different types, because of the usefulness of these traits. You will frequently encounter these traits when working with Rust.

These traits provide a standard way to convert between types, and is sometimes used by the compiler to automatically convert types on your behalf.

As a general rule, you only need to implement the `From` trait, and almost never `Into`. The `Into` trait is the reciprocal of `From`, and will be derived automatically by the compiler. There is one exception to this rule: versions of Rust prior to 1.41 had slightly stricter rules which didn't allow implementing `From` when the conversion destination was an external type.

`From` is preferred because it doesn't require specifying the destination type, resulting in slightly simpler syntax.

The signature for the `From` trait (from the standard library) is as follows:

```
pub trait From<T>: Sized {
    /// Performs the conversion.
    fn from(_: T) -> Self;
}
```

Let's create a very simple `String` wrapper, and implement this trait for our type:

```
struct StringWrapper(String);

impl From<&str> for StringWrapper {
    fn from(other: &str) -> Self {
        Self(other.into())  ❶
    }
}

fn main() {
    println!("{}", StringWrapper::from("Hello, world!").0);
}
```

❶ Returns a copy of the string, wrapped in a new `StringWapper`.

Above, we're allowing conversion from a `&str`, a borrowed string, into a string. To convert the other string into our string, we just call `into()`, which comes from the `Into` trait implemented for `String`. In this example, we use *both* `From` and `Into`.

In practice, you will find yourself needing to convert between types for a variety of reasons. One such case is for handling errors when using `Result`. If you call a function that returns a result, and use the `?` operator within that function, you'll need to provide a `From` implementation if the error type returned by the inner function differs from the error type used by the `Result`.

Consider the following code:

```
use std::{fs::File, io::Read};

struct Error(String);

fn read_file(name: &str) -> Result<String, Error> {
    let mut f = File::open(name)?;
    let mut output = String::new();

    f.read_to_string(&mut output)?;

    Ok(output)
}
```

The code above tries to read a file into a string, and returns the result. We have a custom error type, which just contains a string. The code as it is does not compile:

```
error[E0277]: `?` couldn't convert the error to `Error`
 --> src/main.rs:6:33
  |
5 | fn read_file(name: &str) -> Result<String, Error> {
  |                             --------------------- expected `Error`
  because of this
6 |     let mut f = File::open(name)?;
  |                                 ^ the trait `From<std::io::Error>` is
  not implemented for `Error`
  |
  = note: the question mark operation (`?`) implicitly performs a conversion
  on the error value using the `From` trait
  = note: required by `from`

error[E0277]: `?` couldn't convert the error to `Error`
 --> src/main.rs:9:34
  |
5 | fn read_file(name: &str) -> Result<String, Error> {
  |                             --------------------- expected `Error`
  because of this
...
9 |     f.read_to_string(&mut output)?;
  |                                  ^ the trait `From<std::io::Error>` is
  not implemented for `Error`
  |
  = note: the question mark operation (`?`) implicitly performs a conversion
  on the error value using the `From` trait
  = note: required by `from`
```

To make it compile, we need to implement the `From` trait for `Error` such that the compiler knows how to convert `std::io::Error` into our own custom error. The implementation looks like this:

```
impl From<std::io::Error> for Error {
    fn from(other: std::io::Error) -> Self {
        Self(other.to_string())
    }
}
```

Now, if we compile and run the code, it works as expected.

4.7.1 *TryFrom* and *TryInto*

In addition to the `From` and `Into` traits, there are `TryFrom` and `TryInto`. These traits are nearly identical, except that they are for cases where the type conversion may fail. The conversion methods in these traits return `Result`, whereas with `From` and `Into` there is no way to return an error aside from panicking, which causes the entire program to crash.

4.7.2 Best practices for type conversion using *From* and *Into*

- Implement the `From` trait for types that require conversion to and from other types
- Avoid writing custom conversion routines, and instead rely on the well-known traits where possible

4.8 Handling FFI compatibility with Rust's types

You may occasionally need to call functions from non-Rust libraries (or vice versa), and in many cases that requires modeling C structs in Rust. To do this, you must use Rust's foreign function interface features (usually referred to as *FFI*). Rust's structs are *not* compatible with C structs. In order to make them compatible, you should do the following:

- Structs should be declared with the `#[repr(C)]` attribute, which tells the compiler to pack the struct in a C-compatible representation.
- You should use C types from the `libc` crate, which map directly to C types. Rust types *are not* C types, and you can't always assume they'll be compatible even when you think they're equivalent.

To make this whole process much easier, the Rust team provides a tool called `rust-bindgen`. With `rust-bindgen`, you can generate bindings to C libraries automatically from C headers. Most of the time, you should use `rust-bindgen` to generate bindings, and you can follow the instructions at https://rust-lang.github.io/rust-bindgen/introduction.html to do so.

In some cases I have found I need to call C functions for test purposes or some other reason, and dealing with `rust-bindgen` is not worth the trouble for simple cases. In those cases, the process for mapping C structs to Rust is as follows:

- Copy the C struct definition
- Convert the C types to Rust types
- Implement function interfaces

Following up on the zlib example from chapter 2, let's quickly implement zlib's file struct, which looks like this in C:

```
struct gzFile_s {
    unsigned have;
    unsigned char *next;
    z_off64_t pos;
};
```

The corresponding Rust struct, after conversion, would look like this:

```
#[repr(C)]   ❶
struct GzFileState {   ❷
    have: c_uint,
    next: *mut c_uchar,
    pos: i64,
}
```

❶ Instructs `rustc` to align the memory in this struct as a C compiler would, for compatibility with C.

❷ A C struct, representing a zlib file state, as defined in `zlib.h`.

Putting it all together, you can call C functions from zlib with the struct that zlib expects:

```
type GzFile = *mut GzFileState;

#[link(name = "z")]   ❶
extern "C" {   ❷
    fn gzopen(path: *const c_char, mode: *const c_char) -> GzFile;   ❷
    fn gzread(file: GzFile, buf: *mut c_uchar, len: c_uint) -> c_int;   ❷
    fn gzclose(file: GzFile) -> c_int;   ❷
    fn gzeof(file: GzFile) -> c_int;   ❷
}

fn read_gz_file(name: &str) -> String {
    let mut buffer = [0u8; 0x1000];
    let mut contents = String::new();
    unsafe {
        let c_name = CString::new(name).expect("CString failed");   ❸
        let c_mode = CString::new("r").expect("CString failed");
        let file = gzopen(c_name.as_ptr(), c_mode.as_ptr());
        if file.is_null() {
            panic!(
                "Couldn't read file: {}",
                std::io::Error::last_os_error()
            );
        }
        while gzeof(file) == 0 {
            let bytes_read = gzread(
                file,
                buffer.as_mut_ptr(),
                (buffer.len() - 1) as c_uint,
            );
            let s = std::str::from_utf8(&buffer[..(bytes_read as usize)])
                .unwrap();
            contents.push_str(s);
        }
        gzclose(file);
    }

    contents
}
```

❶ Instructs `rustc` that these functions belong to the external `z` library.

❷ External zlib functions, as defined in `zlib.h`.

❸ Converts a Rust UTF-8 string into an ASCII C string, raising an error if there's a failure.

The `read_gz_file()` will open a gzipped file, read its contents, and return them as a string.

4.9 Summary

- `str` is Rust's stack-allocated UTF-8 string type. A `String` is a heap-allocated UTF-8 string, based on `Vec`.
- A `&str` is a string slice, which can be borrow from both a `String` and `&'static str`.
- `Vec` is a heap-allocated, resizable sequence of values, allocated in a contiguous region of memory. In most cases, you should use a `Vec` when modeling a sequence of values.
- `HashMap` is Rust's standard hash map container type, which is suitable for most uses requiring constant-time lookups from a key.
- Rust also has `VecDeque`, `LinkedList`, `BTreeMap`, `HashSet`, `BTreeSet`, and `BinaryHeap` within its collections library.
- Structs are composable containers, and Rust's primary building block. They are used to store state and implement methods that operate on that state.
- Enums are a special variant type in Rust, and can also emulate the behaviour of `enum` from languages like C, C++, C#, and Java
- Implementations of many standard traits can be derived using the `#[derive]` attribute. If needed, you can manually implement these traits, but most of the time the automatically derived implementations are sufficient.

5 Working with memory

This chapter covers

- Learning about heap and stack-based memory management details in Rust
- Understanding Rust's ownership semantics
- Using reference counted pointers
- Effectively utilizing smart pointers
- Implementing custom allocators for specific use cases

In chapter 4 we discussed Rust's data structures, but to complete our understanding we also need to discuss memory management and how it works with Rust's data structures. The core data structures provide nice abstractions for managing memory allocation and deallocation, but some applications may require more advanced features that require custom allocators, reference counting, smart pointers, or system-level features that are outside the scope of the Rust language.

It's possible to effectively use Rust without having a deep understanding of memory management, but there are many cases in which it's quite beneficial to know what's going on beneath the hood, so to speak. In this chapter we'll get into the details of Rust's memory management.

5.1 Memory management: heap and stack

Rust has very powerful and fine-grained memory management semantics. You may find, when you're new to Rust, that it seems somewhat opaque at first. For example, when you use a String or a Vec, you likely aren't thinking too much about how the memory is allocated. In some ways this is similar to scripting languages such as Python or Ruby where memory management is largely abstracted away, and rarely something you need to think about.

Under the hood, Rust's memory management is not too different from languages like C or C++. In Rust, however, the language tries to keep memory management out of your way until you *need* to worry about memory management. And when you do, the language provides the tools you'll need to dial the complexity up or down depending on what you're trying to accomplish.

Let's quickly review the differences between the *heap* and the *stack*.

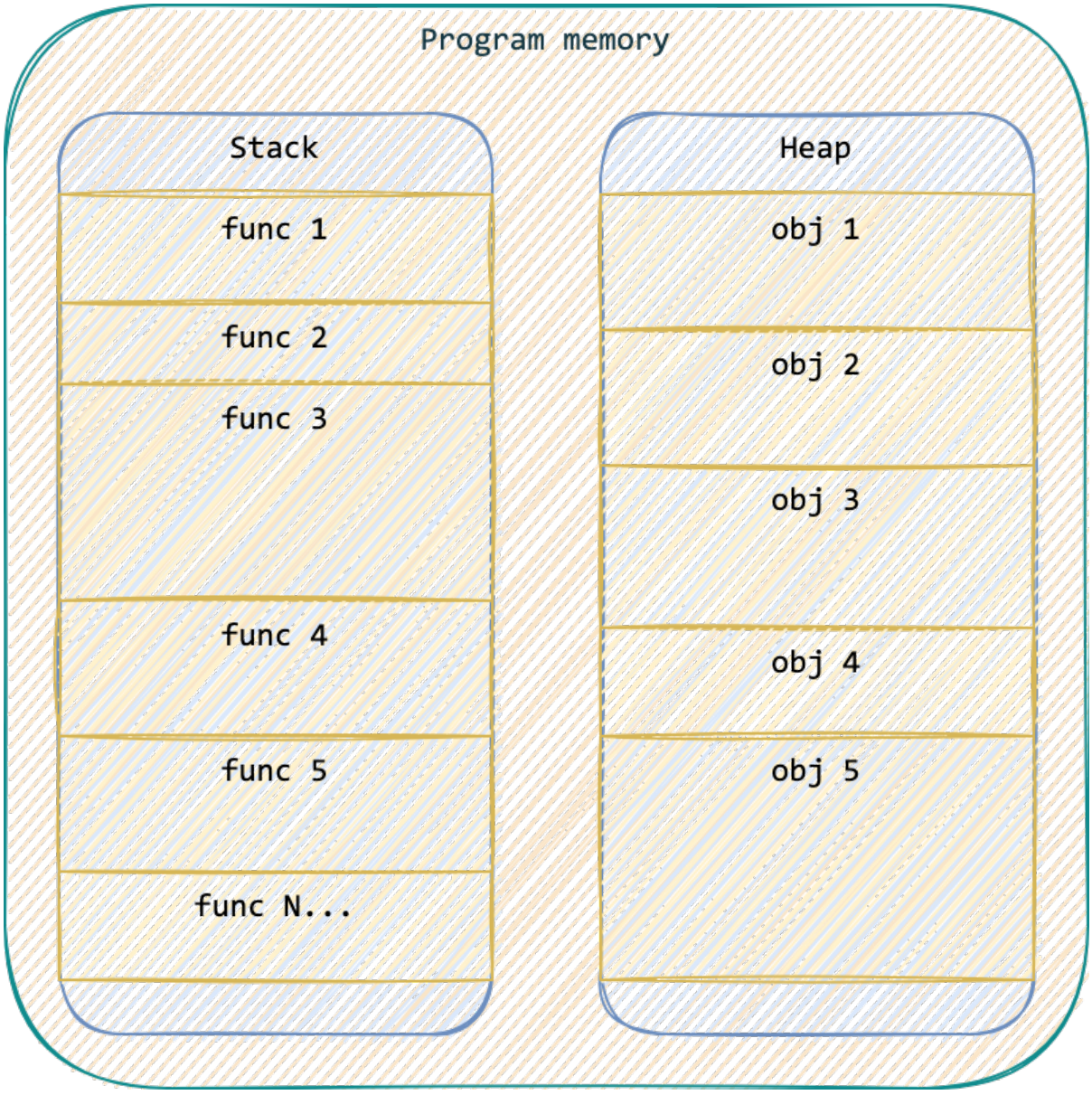

Figure 5.1 Diagram showing example layout of stack and heap

The *heap* is a section of memory for dynamic allocation. This is typically a location in memory reserved for resizable data structures, or anything where the size is only known at runtime. That is not to say you cannot store static data in the heap, however for static data it's usually optimal

to use the stack. The heap is typically managed by the underlying OS or core language libraries, however programmers may–if they choose–implement their own heap. For systems which are memory constrained, such as embedded systems, it's common to write code *without* a heap.

The heap is usually managed by an allocator, and in most cases the operating system, language runtime, or C library provides an allocator (such as `malloc()`). Data in the heap can be thought of as allocated randomly throughout the heap, and can grow and shrink throughout the life of the process.

In Rust, allocating on the heap is accomplished by using any heap-allocated data structure, such as `Vec`, or *`Box`* (`Box` is discussed in more detail later in this chapter):

Listing 5.1 Code listing showing heap-allocated values

```
let heap_integer = Box::new(1);
let heap_integer_vec = vec![0; 100];
let heap_string = String::from("heap string");   ❶
```

❶ As noted in chapter 4, `String` is based on `Vec`, which makes this a heap-allocated string.

The *stack* is a thread-local memory space that is bound to the scope of a function. The stack is allocated using LIFO (last in, first out) order. When a function is entered, the memory is allocated and pushed onto the stack. When a function is exited, the memory is released and popped off the stack. For stack-allocated data, the size needs to be known at compile-time. Allocating memory on the stack is normally much faster than using the heap. There is one stack per thread of execution on operating systems that support it.

The stack is managed by the program itself based on the code generated by the compiler. When a function is entered, a new frame is pushed onto the stack (appended to the end of the stack), and the frame is popped off the stack when leaving the function. As a programmer you don't have to worry about managing the stack, it's handled for you. The stack has some nice properties in that it's fast, and the function call stack can be used as a data structure by making recursive calls; there's no need to worry about memory management.

Listing 5.2 Code listing showing stack-allocated values

```
let stack_integer = 69420;
let stack_allocated_string = "stack string";
```

Many languages obfuscate or abstract away the concepts of stack and heap so you don't have to worry about it. In C and C++, you typically allocate memory on the heap using `malloc()` or the `new` keyword, and simply declare a variable within a function to allocate it on the stack. Java also features the `new` keyword for allocating memory on the heap, however in Java memory is garbage collected and you don't need to manage cleanup of the heap.

In Rust, the stack is managed by the compiler and platform implementation details. Allocating data on the heap, on the other hand, can be customized to suit your needs (we'll discuss custom allocators later in this chapter), which is similar to what you might find in C or C++.

The only types which can be allocated on the stack are primitive types, compound types (such as tuples and structs), `str`, and the container types themselves (but not necessarily their contents).

5.2 Understanding ownership: copies, borrowing, references, and moves

Rust introduces a new programming concept called *ownership*, which is part of what makes Rust different from other languages. Ownership in Rust is where its safety guarantees come from: it's how the compiler knows when memory is in scope, being shared, has gone out of scope, or is being misused. The compiler's *borrow checker* is responsible for enforcing a small set of ownership rules: every value has an owner, there can only be one owner at a time, and when the owner goes out of scope the value is dropped.

If you're already comfortable with Rust's ownership, this section will serve as a review for you, or you can skip it if you wish. On the other hand, if you're still trying to get a handle on ownership in Rust, this section should help clarify those concepts in familiar terms.

Rust's ownership semantics are similar in some ways to C, C++, and Java, except that Rust has no concept of *copy constructors* (which create a copy of an object upon assignment), and you rarely interact with raw pointers in Rust. When you assign the value of one variable to another (i.e., `let a = b;`) it's called a *move*, which is a transfer of ownership (and a value can only have one owner). A move doesn't create a copy *unless* you're assigning a base type (i.e., assigning an integer to another value creates a copy).

Rather than using pointers, in Rust we often pass data around using *references*. In Rust, a reference is created by *borrowing*. Data can be passed into functions by value (which is a move) or by reference. While Rust does have C-like pointers, they aren't something you'll see very often in Rust, except perhaps when interacting with C code.

Borrowed data (i.e., a reference) can either be immutable or mutable. By default, when you borrow data you do so immutably (i.e., you can't modify the data pointed to by the reference). If you borrow with the `mut` keyword, you can obtain a mutable reference which allows you to modify data. You can borrow data immutably simultaneously (i.e., have multiple references to the same data), but you cannot borrow data mutably more than once at a time.

Borrowing is typically done using the `&` operator (or `&mut` to borrow mutably), however you'll sometimes see `as_ref()` or `as_mut()` methods being used instead, which are from the `AsRef` and `AsMut` traits respectively. `as_ref()` and `as_mut()` are often used by container types to

provide access to internal data, rather than obtaining a reference to the container itself (and we'll explore this in more detail later in this chapter).

To clarify these concepts, consider the following code sample:

Listing 5.3 Code listing to demonstrate ownership

```
fn main() {
    let mut top_grossing_films =   ❶
        vec!["Avatar", "Avengers: Endgame", "Titanic"];
    let top_grossing_films_mutable_reference =
        &mut top_grossing_films;   ❷
    top_grossing_films_mutable_reference   ❸
        .push("Star Wars: The Force Awakens");
    let top_grossing_films_reference = &top_grossing_films;   ❹
    println!(
        "Printed using immutable reference: {:#?}",
        top_grossing_films_reference   ❺
    );
    let top_grossing_films_moved = top_grossing_films;   ❻
    println!("Printed after moving: {:#?}", top_grossing_films_moved);   ❼

    // println!("Print using original value: {:#?}", top_grossing_films);   ❽
    // println!(
    //     "Print using mutable reference: {:#?}",
    //     top_grossing_films_mutable_reference   ❾
    // );
}
```

❶ Here we create a mutable `Vec` and populate it with some values

❷ This borrows a mutable reference to the `Vec` above

❸ We can use this mutable reference to modify the data that was borrow in the line above

❹ Now we'll take an immutable reference of the same data, and by doing so the previous mutable reference becomes invalid

❺ Here we print the contents of the `Vec`

❻ This assignment is a move, which transfers ownership of the `Vec`

❼ Here we print the contents of the `Vec` after moving it

❽ The original variable is no longer valid as it has been moved, so this code won't compile

❾ This code also won't compile, because this reference was invalidated when we created the immutable reference

Running the code above produces the following output:

```
Printed using immutable reference: [
    "Avatar",
    "Avengers: Endgame",
    "Titanic",
    "Star Wars: The Force Awakens",
]
Printed after moving: [
    "Avatar",
    "Avengers: Endgame",
    "Titanic",
    "Star Wars: The Force Awakens",
]
```

5.3 Deep copying

You may have encountered the concept of *deep copying* from other languages, such as Python or Ruby. The need for deep copying arises when the language or data structures implement optimizations to prevent copying data, typically through the use of pointers, references, and copy-on-write semantics.

Copies of data structures can either be *shallow* (copying a pointer or creating a reference) or *deep* (copying or cloning all the *values* within a structure, recursively). Some languages perform shallow copies by default when you make an assignment (`a = b`), or call a function. Thus, if you come from languages like Python, Ruby, or JavaScript, you may have needed to occasionally perform an explicit deep copy. Rust doesn't assume anything about your intentions, thus you always need to explicitly instruct the compiler what to do. In other words, the concept of shallow copies does not exist Rust, but rather we have borrowing and references.

Languages that use implicit data references can create undesired side effects, and this may occasionally catch people off-guard or create hard to find bugs. The problem occurs, generally speaking, when you intend to make a *copy*, but the language instead provides a *reference*. Lucky for you, Rust doesn't do any type of implicit data referencing magic, but only so long as you stick to core data structures.

In Rust, the term *cloning* (rather than "copying") is used to describe the process of creating a new data structure and copying (or more correctly, cloning) all the data from the old structure into the new one. The operation is typically handled through the `clone()` method, which comes from the `Clone` trait, and can be automatically derived using the `#[derive(Clone)]` attribute (traits and deriving traits is discussed in more detail in chapters 8 & 9). Many data structures in Rust come with the `Clone` trait implemented for you, so you can usually count on `clone()` being available.

Consider the following code:

Listing 5.4 Code listing to demonstrate `clone()`

```
fn main() {
    let mut most_populous_us_cities =
        vec!["New York City", "Los Angeles", "Chicago", "Houston"];
    let most_populous_us_cities_cloned = most_populous_us_cities.clone();  ❶
    most_populous_us_cities.push("Phoenix");  ❷
    println!("most_populous_us_cities = {:#?}", most_populous_us_cities);
    println!(
        "most_populous_us_cities_cloned = {:#?}",
        most_populous_us_cities_cloned  ❸
    );
}
```

❶ Here we clone the original `Vec`

❷ We'll add a new city to the list in the original `Vec`

❸ When the cloned `Vec` is printed, it won't output "Phoenix" because it's an entirely distinct structure

Running the code above prints the following:

```
most_populous_us_cities = [
    "New York City",
    "Los Angeles",
    "Chicago",
    "Houston",
    "Phoenix",
]
most_populous_us_cities_cloned = [
    "New York City",
    "Los Angeles",
    "Chicago",
    "Houston",
]
```

The `Clone` trait, when derived, operates recursively. Thus, calling `clone()` on any top-level data structure, such as a `Vec`, is sufficient to create a deep copy of the contents of the `Vec` provided they all implement `Clone`. Deeply nested structures can be easily cloned without needing to do anything beyond ensuring they implement the `Clone` trait.

5.4 Avoiding copies

There are certain cases where, perhaps unintentionally, data structures can wind up being cloned or copied more often than needed. This can happen in string processing for example, where many copies of a string are made repeatedly within algorithms which scan, mutate, or otherwise handle some arbitrary set of data.

One downside to `Clone` is that it can be *too easy* to copy data structures, and you may end up with many copies of the same same data if applied too liberally. For most intents and purposes this is a non-issue, and you're unlikely to have problems until you start operating on very large sets of data.

Many core library functions in Rust return copies of objects, as opposed to modifying them in place. This is, in most cases, the preferred behaviour: it helps maintain immutability of data which makes it easier to reason about how algorithms behave, at the cost of duplicating memory, perhaps only temporarily. To illustrate, let's examine a few string operations from Rust's core library:

Table 5.1 Examining Rust core string functions for copies

Function	Description	Copies?	Algorithm	Identified by?
`pub fn replace<'a, P>(&'a self, from: P, to: &str) String where P: Pattern<'a>,`	Replaces all matches of a pattern with another string	Yes	Creates a new string, pushes updated contents into new string, returning the new string and leaving the original string untouched	`self` parameter is an immutable reference; function returns owned `String`
`pub fn to_lowercase(&self) String`	Returns the lowercase equivalent of this string slice, as a new String	Yes	Creates a new string, copies each character to the new string, converting uppercase characters to lowercase characters	`self` parameter is an immutable reference; function returns owned `String`
`pub fn make_ascii_lowercase(&mut self)`	Converts this string to its ASCII lower case equivalent in-place	No	Iterates over each character, applying a lowercase conversion on uppercase ASCII characters	Function takes a mutable `self` reference, modifying the memory in place
`pub fn trim(&self) &str`	Returns a string slice with leading and trailing whitespace removed	No	Uses a double-ended searcher to find the start and end of a substring without whitespace, returning a slice representing the trimmed result	Function returns reference, not an owned string

You'll notice a pattern above, which is that you can often identify whether an algorithm creates a copy based on whether the function modifies the source data in place, or returns a new copy.

There's one more case to illustrate, which is what I call the *pass through*. Consider the following:

```
fn lowercased(s: String) -> String {
    s.to_lowercase()   ❶
}

fn lowercased_ascii(mut s: String) -> String {
    s.make_ascii_lowercase();   ❷
    s
}
```

❶ A copy is created here, inside `to_lowercase()`, and the new string is returned.

❷ The string is passed through directly, with the memory modified in place. The ownership is passed back to the caller by returning the same owned object, as the `make_ascii_lowercase()` function operates in-place.

In the code above, the first function, `lowercased()`, takes an owned string but returns a new

copy of that string, by calling `to_lowercase()`. The second function takes a mutable owned string, and returns a lowercased version using the *in-place* version (which only works on ASCII strings).

To summarize:

- Functions that take immutable references and return a reference or slice are unlikely to make copies (e.g., `fn func(&self) &str`)
- Functions that take a reference (i.e., `&`) and returns an owned object, it *may* be creating a copy (e.g., `fn func(&self) String`)
- Functions that take a *mutable* reference (i.e., `&mut`), it may be modifying data in place (e.g., `fn func(&mut self)`)
- Functions that take an owned object and return an owned object of the same type are *probably* making a copy (e.g., `fn func(String) String`)
- Functions that take a mutable owned object and return an owned object of the same type may not be making a copy (e.g., `fn func(mut String) String`)

As a general rule, you *should* examine documentation and source code when you're unsure as to whether functions make copies, operate in place, or merely pass ownership. Rust's memory semantics do make it relatively easy to reason about how algorithms operate on data merely by examining the inputs and outputs, but this only works provided the functions being called follow these patterns. For cases where you have serious performance concerns, you should closely examine the underlying algorithms.

5.5 To box or not to box: smart pointers

Rust's *`Box`* is a type of smart pointer, which we briefly discussed in chapter 4. `Box` is a bit different from smart pointers in languages like C++, as its main purpose is providing a way to allocate data on the heap. In Rust the 2 main ways to allocate data on the heap are by using `Vec`, or by using `Box`. `Box` is quite limited in terms of its capabilities: it only handles allocation and deallocation of memory for the object it holds, without providing much else in terms of features, but that's by design. `Box` is still very useful, and it should be the first thing you reach for in cases where you need to store data on the heap (aside from using a `Vec`).

YOUR TURN **Since a `Box` cannot be empty (except in certain situations which are best left unexplored), we often hold a `Box` within an *`Option`*, in any case where boxed data might not be present.**

If the data or object is optional, you should put your `Box` in an `Option`. Optional types (or *maybe* types) aren't unique to Rust, you may have encountered them from Ada, Haskell, Scala, or Swift, to name a few languages which have them. Optionals are a kind of *monad*—a functional design pattern whereby you wrap values that shouldn't have unrestricted access in a function. Rust provides some syntax sugar to make working with optionals more pleasant.

You will see `Option` used frequently in Rust; if you haven't encountered optionals before, you can think of them as a way to safely handle null values (such as pointers). Rust doesn't have null pointers (excluding unsafe code), but it *does* have `None`, which is functionally equivalent to a null pointer, without the safety problems.

The cool thing about `Box` and `Option` is that when both are used together it's nearly impossible to have runtime errors (such as null pointer exceptions) due to invalid, uninitialized, or doubly-freed memory. There is one caveat, however: heap allocations may fail. Handling this situation is tricky, outside the scope of this book, and somewhat dependent on the operating system and its settings.

One common cause of allocation failures is the system running out of free memory, and handling this (if you choose to handle it at all) is largely dependent on the application. Most of the time the expected result for failed memory allocations is for the program to "fail fast" and exit with an out of memory error (OOM), which is almost always the default behaviour (i.e., what will happen if you as a developer don't handle OOM errors). You have likely encountered such situations yourself. Some notable applications that provide their own memory management features are web browsers, which often have their own built-in task managers and memory management much like the OS itself. If you're writing mission critical software, such as a database or online transaction processing system, you may want to handle memory allocation failures gracefully.

In specific situations where you suspect allocation *might* fail, `Box` provides the `try_new()` method, which returns a *`Result`*, which–like an `Option`–may be in either a success or failure state. The default `new()` method of `Box` will create a panic in case allocation fails, which will cause your program to crash. In *most* cases, crashing is the best way to handle failed allocations. Alternatively, you can catch allocation failures within a custom allocator (which is discussed later in this chapter).

YOUR TURN **To better understand `Option` and `Result`, try implementing them yourself using an enum. In Rust, creating and using your own optionals is trivial with enums and pattern matching.**

To illustrate the use of `Box`, consider a basic singly-linked list in Rust:

Listing 5.5 Code listing for a basic singly-linked list in Rust

```
struct ListItem<T> {
    data: Box<T>,   ❶
    next: Option<Box<ListItem<T>>>,   ❷
}

struct SinglyLinkedList<T> {
    head: ListItem<T>,   ❸
}

impl<T> ListItem<T> {
    fn new(data: T) -> Self {
        ListItem {
            data: Box::new(data),   ❹
            next: None,   ❺
        }
    }
    fn next(&self) -> Option<&Self> {   ❻
        if let Some(next) = &self.next {   ❼
            Some(next.as_ref())   ❽
        } else {
            None
        }
    }
    fn mut_tail(&mut self) -> &mut Self {
        if self.next.is_some() {   ❾
            self.next.as_mut().unwrap().mut_tail()   ❿
        } else {
            self   ⓫
        }
    }
    fn data(&self) -> &T {
        self.data.as_ref()   ⓬
    }
}

impl<T> SinglyLinkedList<T> {
    fn new(data: T) -> Self {
        SinglyLinkedList {
            head: ListItem::new(data),   ⓭
        }
    }
    fn append(&mut self, data: T) {
        let mut tail = self.head.mut_tail();   ⓮
        tail.next = Some(Box::new(ListItem::new(data)));   ⓯
    }
    fn head(&self) -> &ListItem<T> {
        &self.head   ⓰
    }
}
```

❶ Data is boxed within each list item. The data field can’t be empty or null.

❷ The *next* pointer is optional: we don’t know if there’s a subsequent element in the list, so we place a `Box` in an `Option`, pointing to the next item in the list.

❸ The struct for the list itself only contains the head: we don’t bother boxing the head, because it must always be present.

❹ The new data is moved into the new list item by placing it into a `Box`; the data is allocated on the heap, and the compiler will sort out the details on getting the data into the target location, as it may need to be moved from stack onto the heap.

❺ The `next` pointer is initialized as `None`, because new elements don't know where they are in the list yet. Plus, this implementation doesn't have an insert operation, only *append*.

❻ The `next()` method on each item returns an optional reference to the next item, if it exists. This function exists to help unwrap the nested references for the sake of simplifying the code.

❼ We're using an `if let …` construct to check if the `next` pointer points to anything before trying to dereference it.

❽ Here we dereference the `next Box` pointer, convert it to an ordinary reference, and return that inside an option. This is merely to make the code a little easier to use, as the alternative is to return a reference to a `Box`, rather than the object directly. This could also be written as `Some(&*next)`.

❾ We can't use the `if let …` construct here, because we need to borrow `self.next`. The compiler won't let us borrow the same object twice, so we merely check to see if `next` is present without creating a borrowed object.

❿ Because our `next` pointer is a `Box` within an `Option`, we need to unwrap *both* the `Option` and the `Box`. Additionally, we need to return a *mutable* reference, so we have to get an inner mutable reference from the `Option` (i.e., a `Option<&mut T>` instead of `&mut Option<T>`) with `as_mut()`, then we unwrap the result with `unwrap()`, and rely on an implicit dereference of `Box<T>` to return a mutable reference to the underlying item (`&mut ListItem<T>`).

⓫ If there's no `next` element, *this* item is the tail; just return `self`.

⓬ Here we provide a convenient method to access the data directly, which just calls the `as_ref()` method on the boxed data. This isn't necessary, it just makes the list interface a little cleaner.

⓭ Creating a new list requires a first element, we won't bother supporting empty lists, but it could be a fun exercise for the reader. To support an empty list, you can make the head element optional and update the code accordingly.

⓮ To append a new element, first we fetch the last element in the list. Our list can never be empty, so we don't even bother checking the tail exists.

⓯ We add our new element to the tail item's `next` pointer, and the new element becomes the new tail.

⓰ Here we provide a method to get the head element for convenience. This isn't needed in the example here, but if you were developing a data structure you'd want to provide convenient accessors (such as `head()` and `tail()`) rather than making the individual struct elements public.

There's a lot to unpack in the linked list example. For someone new to Rust, implementing a linked list is one of *the best* ways to learn about Rust's unique features. The example above provides some nice features, and it's *safe*. The list will never be empty, invalid, or contain null pointers. This is a really powerful feature of Rust, and it's only possible thanks to Rust's rules about object ownership.

We can test the linked list we just created using the following code:

```
fn main() {
    let mut list = SinglyLinkedList::new("head");   ❶
    list.append("middle");   ❶
    list.append("tail");   ❶
    let mut item = list.head();   ❷
    loop {   ❸
        println!("item: {}", item.data());   ❹
        if let Some(next_item) = item.next() {   ❺
            item = next_item;
        } else {
            break;   ❻
        }
    }
}
```

❶ Creates a new linked list of strings, with a head element, and then we add a middle and tail element

❷ Gets a reference to the head of the list

❸ Loop until we've visited every item in the list

❹ Print the value of each item

❺ Fetches the next item in the list using an `if let` statement, which unwraps the `Option`

❻ We terminate the loop with a `break` when we've reached the end of the list, which we know when the next item is `None`

Running the code above produces the following output:

```
item: head
item: middle
item: tail
```

Before moving on to the next sections, I suggest you take some time to understand the singly-linked list in Rust. Try implementing it yourself from scratch, and refer back to the example provided as needed. In the next section we'll do a more advanced version of the linked list. If you care to read on anyway, it may be worth revisiting this exercise if you want to get a better handle on Rust's memory management once you have a good understanding of the overall language.

Lastly: in practice, you'll likely never need to implement your own linked list in Rust. Rust's core library provides `std::collections::LinkedList` in case you *really* want a linked list. For most cases, however, just use a `Vec`. Additionally, the example provided here is not optimized.

5.6 Reference counting

In the previous section we talked about `Box`, which is a useful–but very limited–smart pointer, and something you'll encounter often. Notably, a `Box` cannot be shared. That is to say, you can't have 2 separate boxes pointing to the same data in a Rust program; `Box` owns its data, and doesn't allow more than one borrow at a time. This is, for the most part, a feature (or anti-feature) worth being excited about. However, there are cases in which you *do* want to share data: perhaps across threads of execution, or by storing the same data in multiple structures in order to address it differently (such as a `Vec` and a `HashMap`).

In cases where `Box` doesn't cut it, what you're probably looking for are *reference counted* smart pointers. Reference counting is a common technique in memory management to avoid keeping track of how many copies of a pointer exist, and when there are no more copies the memory is released. The implementation usually relies on keeping a static counter of the number of copies of a given pointer, and incrementing the counter every time a new copy is made. When a copy is destroyed, the counter is decremented. If the counter ever reaches zero, the memory can be released because that means there are no more copies of the pointer and thus the memory is no longer in use or accessible.

YOUR TURN **Implementing a reference-counted smart pointer is a fun exercise to do on your own, however it's a bit tricky in Rust and requires the use of raw (i.e., unsafe) pointers. If you find the linked list exercise too easy, try making your own reference-counted smart pointer.**

Rust provides 2 different reference counted pointers:

- `Rc`: a single-threaded reference-counted smart pointer, which enables shared ownership of an object
- `Arc`: a multi-threaded reference-counted smart pointer, which enables shared ownership of objects across threads

SIDEBAR **Single vs multi-threaded objects in Rust**

Many programming languages distinguish between functions or objects that can be used across threads as thread safe versus unsafe. In Rust, this distinction doesn't quite map directly, as everything is safe by default. Instead, some objects can be moved or synchronized across threads, and others can't. This behaviour comes from whether an object implements the `Send` and `Sync` traits, which we discuss in more detail in chapter 6.

In the case of `Rc` and `Arc`, `Rc` doesn't provide `Send` or `Sync` (in fact, `Rc` explicitly marks these traits as not implemented), so `Rc` can only be used in a single thread. `Arc`, on the other hand, implements both `Send` and `Sync`, thus it can be used in multi-threaded code.

`Arc` in particular uses atomic counters, which are platform dependent and usually implemented at the operating system or CPU level. Atomic operations are more costly than regular arithmetic, so only use `Arc` when you need atomicity.

It's important to note that so long as you aren't using the `unsafe` keyword to bypass language rules, Rust code is always safe. Getting it to compile, on the other hand, can be quite a challenge when you don't understand Rust's unique patterns and jargon.

In order to use reference-counted pointers effectively, we also need to introduce another concept in Rust called *interior mutability*. Interior mutability is something you may need when Rust's borrow checker doesn't provide enough flexibility with mutable references. If this sounds like an escape hatch, then pat yourself on the back for being an astute reader because it *is* an escape hatch. But worry not, it doesn't break Rust's safety contracts, and still allows you to write safe code.

To enable interior mutability, we need to introduce 2 special types in Rust: `Cell` and `RefCell`. If you're new to Rust, you probably haven't encountered these yet, and it's unlikely you would bump into them under normal circumstances. In most cases, you'll want to use `RefCell` rather than `Cell`, as `RefCell` allows us to borrow references, whereas `Cell` moves values in and out of itself (which is probably *not* the behaviour you want most of the time).

Another way to think about `RefCell` and `Cell` is that they allow you to provide the Rust compiler with more information about *how* you want to borrow data. The compiler is quite good, but it's limited in terms of flexibility, and there are some cases where perfectly safe code won't compile because the compiler doesn't understand what you're trying to do (regardless of how correct it might be).

You shouldn't need `RefCell` or `Cell` very often; if you find yourself trying to use these to get around the borrow checker, you might need to rethink what you're doing. They are mainly

needed for specific cases, such as containers and data structures that hold data that needs to be accessed mutably.

One limitation of `Cell` and `RefCell` is that they're only for single-threaded applications. In the case where you require safety across threads of execution, you can use *`Mutex`* or *`RwLock`*, which provide the same feature to enable interior mutability, but can be used across threads. These would typically be paired with `Arc` rather than `Rc` (we'll explore concurrency in more detail in chapter 10).

Let's update the linked list example from the previous section to use `Rc` and `RefCell` instead of `Box`, which gives us more flexibility. Notably, we can now make our singly-linked list *doubly* -linked list. This isn't possible using `Box` because it doesn't allow shared ownership.

Listing 5.6 Code listing of a doubly-linked list using `Rc`, `RefCell`, and `Box`

```
use std::cell::RefCell;
use std::rc::Rc;

struct ListItem<T> {
    prev: Option<ItemRef<T>>,   ❶
    data: Box<T>,   ❷
    next: Option<ItemRef<T>>,
}

type ItemRef<T> = Rc<RefCell<ListItem<T>>>;   ❸

struct DoublyLinkedList<T> {
    head: ItemRef<T>,
}

impl<T> ListItem<T> {
    fn new(data: T) -> Self {
        ListItem {
            prev: None,
            data: Box::new(data),   ❹
            next: None,
        }
    }
    fn data(&self) -> &T {
        self.data.as_ref()
    }
}

impl<T> DoublyLinkedList<T> {
    fn new(data: T) -> Self {
        DoublyLinkedList {
            head: Rc::new(RefCell::new(ListItem::new(data))),
        }
    }
    fn append(&mut self, data: T) {
        let tail = Self::find_tail(self.head.clone());   ❺
        let new_item = Rc::new(RefCell::new(ListItem::new(data)));   ❻
        new_item.borrow_mut().prev = Some(tail.clone());   ❼
        tail.borrow_mut().next = Some(new_item);   ❽
    }
    fn head(&self) -> ItemRef<T> {
        self.head.clone()
    }
    fn tail(&self) -> ItemRef<T> {
        Self::find_tail(self.head())
    }
    fn find_tail(item: ItemRef<T>) -> ItemRef<T> {
        if let Some(next) = &item.borrow().next {   ❾
            Self::find_tail(next.clone())   ❿
        } else {
            item.clone()   ⓫
        }
    }
}
```

❶ We've added a pointer to the *previous* item in the list.

❷ The data is still kept in a `Box`; we don't need to use an `Rc` here because we're not sharing ownership of the data, only the pointers to nodes in the list.

❸ This type alias helps keep the code clean.

❹ Data is moved into a `Box` here.

❺ First we need to find the pointer to the tail item in the list.

❻ Creates a pointer for the new item we're about to append.

❼ We'll update the `prev` pointer in the new item to point to the previous tail.

❽ Update the `next` pointer of the previous tail to point to the new tail, which is the newly inserted item.

❾ Checks if the `next` pointer is empty, and continues searching recursively if not.

❿ We clone the next pointer and return it, continuing the search.

⓫ If the `next` pointer is empty, we're at the end (or tail) of the list. Returns the current item pointer after cloning it.

This version of the linked list looks quite different from the previous version. Introducing `Rc` and `RefCell` adds some complexity, but provides us with a lot more flexibility. We'll revisit this example again later in the book as we explore more language features.

To summarize, `Rc` and `Arc` provide reference-counted pointers, but to access inner data mutably you'll need to use an object such as `RefCell` or `Cell` (and for multi-threaded applications, `Mutex` or `RwLock`).

5.7 Clone on write

Earlier in this chapter we discussed *avoiding* copies. However, there are cases in which you want to *prefer* making copies of data, rather than ever mutating data in place. This pattern has some very nice features, especially if you prefer functional programming patterns. You may not have heard of *clone on write* before, but you're probably familiar with *copy on write*.

Copy on write is a design pattern where data is never mutated in place, but rather, any time data needs to be changed it's copied to a new location, mutated, and then a reference to the new copy of data is returned. Some programming languages enforce this pattern as a matter of principle, such as in Scala where data structures are classified as either *mutable* or *immutable*, and all the immutable structures implement copy on write. A very popular JavaScript library, Immutable.js, is based entirely on this pattern with all data structure mutations resulting in a new copy of the data. Building data structures based on this pattern makes it much easier to reason about how data is handled within programs.

For example, with a copy on write list or array, the append operation would return a new list with all the old elements, plus the new element appended, while leaving the original list of items in tact. The programmer assumes the compiler can handle optimizations and cleaning up of old data.

In Rust, this pattern is referred to as clone on write, as it depends on the `Clone` trait. `Clone`

differs from `Copy` in Rust in order to distinguish from an implicit bitwise copy (i.e., literally copying the bytes of an object to a new memory location) versus an *explicit* copy, meaning–in the case of `Clone`–calling the `clone()` method on an object. The `Clone` trait is normally implemented automatically using `#[derive(Clone)]`, but it can be implemented manually for special cases. We'll explore traits in more detail in chapter 8 and 9.

Rust provides 3 smart pointers to help with implementing clone on write:

- `Cow`: an enum-based smart pointer that provides convenient semantics
- `Rc` and `Arc`: both reference-counted smart pointers provide clone on write semantics with the `make_mut()` method. `Rc` is the single-threaded version, and `Arc` is the multi-threaded version.

Let's look at the type signature for `Cow`:

Listing 5.7 Snippet of `Cow` definition from Rust standard library

```
pub enum Cow<'a, B> where
    B: 'a + ToOwned + ?Sized,  {
    Borrowed(&'a B),
    Owned(<B as ToOwned>::Owned),
}
```

`Cow` is an enum that can contain *either* a borrowed variant, or an owned variant. For the owned variant, it behaves a lot like `Box`, except that with `Cow` the data is not necessarily allocated on the heap. If you want heap-allocated data with `Cow`, you'll need to use a `Box` within `Cow`, or use `Rc` or `Arc` instead. Rust's clone on write feature is also not a language level feature: you need to explicitly use the `Cow` trait.

To demonstrate the use of `Cow`, let's update the singly-linked list example so that the data structure becomes immutable. First, let's examine the listing 5.8, which aside from adding `#[derive(Clone)]`, isn't too different from the previous version:

Listing 5.8 Code listing of `ListItem` for singly-linked list using `Cow`

```
#[derive(Clone)]   ❶
struct ListItem<T>
where
    T: Clone,
{
    data: Box<T>,
    next: Option<Box<ListItem<T>>>,
}

impl<T> ListItem<T>
where
    T: Clone,
{
    fn new(data: T) -> Self {
        ListItem {
            data: Box::new(data),
            next: None,
        }
    }
    fn next(&self) -> Option<&Self> {
        if let Some(next) = &self.next {
            Some(&*next)
        } else {
            None
        }
    }
    fn mut_tail(&mut self) -> &mut Self {
        if self.next.is_some() {
            self.next.as_mut().unwrap().mut_tail()
        } else {
            self
        }
    }
    fn data(&self) -> &T {
        self.data.as_ref()
    }
}
```

❶ We derive the `Clone` trait for both structs. `Cow` depends on the behaviour of the `Clone` trait.

Next, let's look at listing 5.9, which shows the usage of `Cow` in our list:

Listing 5.9 Code listing of `SinglyLinkedList` for singly-linked list using `Cow`

```
#[derive(Clone)]
struct SinglyLinkedList<'a, T>
where
    T: Clone,
{
    head: Cow<'a, ListItem<T>>,   ❶
}

impl<T> ListItem<T>
where
    T: Clone,
{
    fn new(data: T) -> Self {
        ListItem {
            data: Box::new(data),
            next: None,
        }
    }
    fn next(&self) -> Option<&Self> {
        if let Some(next) = &self.next {
            Some(&*next)
        } else {
            None
        }
    }
    fn mut_tail(&mut self) -> &mut Self {
        if self.next.is_some() {
            self.next.as_mut().unwrap().mut_tail()
        } else {
            self
        }
    }
    fn data(&self) -> &T {
        self.data.as_ref()
    }
}

impl<'a, T> SinglyLinkedList<'a, T>
where
    T: Clone,
{
    fn new(data: T) -> Self {
        SinglyLinkedList {
            head: Cow::Owned(ListItem::new(data)),   ❷
        }
    }
    fn append(&self, data: T) -> Self {   ❸
        let mut new_list = self.clone();
        let mut tail = new_list.head.to_mut().mut_tail();   ❹
        tail.next = Some(Box::new(ListItem::new(data)));
        new_list
    }
    fn head(&self) -> &ListItem<T> {
        &self.head
    }
}
```

❶ The `head` pointer is stored within a `Cow`. We have to include lifetime specifier for the struct so the compiler knows that the struct and the `head` parameter have the same lifetime.

❷ Here we initialize the list with the `head` pointer.

❸ The append signature has changed such that it no longer requires a mutable `self`, and instead it returns an entirely new linked list.

❹ The call to `to_mut()` triggers the clone on write, which happens recursively, by obtaining a mutable reference to the `head`.

5.8 Custom allocators

In some cases you may find yourself needing to customize memory allocation behaviour. Some example cases of this are:

- Embedded systems, which are highly memory constrained or lack an operating system
- Performance critical applications that required optimized memory allocation, including custom heap managers such as jemalloc[18] or TCMalloc[19]
- Applications with strict security or safety requirements, where you may want to protect memory pages using the `mprotect()` and `mlock()` system calls, for example
- Some library or plugin interfaces may require special allocators when handing off data to avoid memory leaks; this is quite common when working across language boundaries (i.e., integrating between Rust and a garbage collected language)
- Implementing custom heap management, such as memory usage tracking from *within* your application

By default, Rust will use the standard system implementation for memory allocation, which on most systems is the `malloc()` and `free()` functions provided by the system's C library. This behaviour is implemented by Rust's *global allocator*. The global allocator can be overridden for an entire Rust program using the `GlobalAlloc` API, and individual data structures can be overridden using custom allocators with the `Allocator` API.

> **NOTE** **The `Allocator` API in Rust is a nightly-only feature as of the time of writing. For more details on the status of this feature, refer to https://github.com/rust-lang/rust/issues/32838. You can still use the `GlobalAlloc` API in stable Rust.**

Even if you never need to write your own allocator (most people are unlikely to need a custom allocator), it's worth getting a feel for the allocator interface to have a better understanding of Rust's memory management. In practice you're unlikely to ever need to worry about allocators except in special circumstances such as those mentioned above.

5.8.1 Writing a custom allocator

Let's explore writing a custom `Allocator`, which we'll use with a `Vec`. Our allocator will simply call the `malloc()` and `free()` functions. To start, let's examine the `Allocator` trait, as defined in the Rust standard library at https://doc.rust-lang.org/std/alloc/trait.Allocator.html. The trait is as follows:

Listing 5.10 Code listing for `Allocator` trait, from Rust standard library

```
pub unsafe trait Allocator {
    fn allocate(&self, layout: Layout)
      -> Result<NonNull<[u8]>, AllocError>;   ❶
    unsafe fn deallocate(&self, ptr: NonNull<u8>, layout: Layout);   ❶

    fn allocate_zeroed(   ❷
        &self,
        layout: Layout
    ) -> Result<NonNull<[u8]>, AllocError> { ... }
    unsafe fn grow(   ❷
        &self,
        ptr: NonNull<u8>,
        old_layout: Layout,
        new_layout: Layout
    ) -> Result<NonNull<[u8]>, AllocError> { ... }
    unsafe fn grow_zeroed(   ❷
        &self,
        ptr: NonNull<u8>,
        old_layout: Layout,
        new_layout: Layout
    ) -> Result<NonNull<[u8]>, AllocError> { ... }
    unsafe fn shrink(   ❷
        &self,
        ptr: NonNull<u8>,
        old_layout: Layout,
        new_layout: Layout
    ) -> Result<NonNull<[u8]>, AllocError> { ... }
    fn by_ref(&self) -> &Self { ... }   ❷
}
```

❶ Required methods.

❷ Optional methods, with default implementations provided.

To implement an allocator, we only need to provide 2 methods: `allocate()` and `deallocate()`. These are analogous to `malloc()` and `free()`. The other methods are provided for cases where you wish to optimize allocation further. The C-equivalent call for `allocated_zeroed()` would be `calloc()`, whereas for the grow and shrink functions you'd use `realloc()`.

> **NOTE** **You may notice the *`unsafe`* keyword on some of the `Allocator` trait's methods. Allocating and deallocating memory nearly always involves unsafe operations in Rust, which is why these methods are marked as unsafe.**

Rust provides default implementations for the optional methods in the `Allocator` trait. In the case of growing and shrinking, the default implementation will simply allocate new memory,

copy all the data, then deallocate old memory. For allocating zeroed data, the default implementation calls `allocate()` and writes zeroes to all the memory locations.

Let's begin by writing an allocator that passes through to the global allocator:

Listing 5.11 Code listing for a pass-through allocator

```
#![feature(allocator_api)]

use std::alloc::{AllocError, Allocator, Global, Layout};
use std::ptr::NonNull;

pub struct PassThruAllocator;

unsafe impl Allocator for PassThruAllocator {
  fn allocate(&self, layout: Layout) -> Result<NonNull<[u8]>, AllocError> {
      Global.allocate(layout)
  }
  unsafe fn deallocate(&self, ptr: NonNull<u8>, layout: Layout) {
      Global.deallocate(ptr, layout)
  }
}
```

NOTE **The code samples for the allocator API are nightly-only, and to compile or run them you need to either use `cargo +nightly …`, or override the toolchain within the project directory with `rustup override set nightly`.**

The code above creates a pass-through allocator, which simply calls the underlying global allocator implementation, with the minimum required code.

To test our allocator:

```
fn main() {
    let mut custom_alloc_vec: Vec<i32, _> =
        Vec::with_capacity_in(10, BasicAllocator);   ❶
    for i in 0..10 {
        custom_alloc_vec.push(i as i32 + 1);
    }
    println!("custom_alloc_vec={:?}", custom_alloc_vec);
}
```

❶ Creates a `Vec` using our custom allocator, initializing the vector with a capacity of 10 items.

Running the code above provides the following output, as expected:

```
custom_alloc_vec=[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

Now, let's change the allocator to call the `malloc()` and `free()` functions directly from the C library instead. The `Layout` struct provides us with the details needed to determine how much memory to allocate using the `size()` method:

Listing 5.12 Code listing for a basic custom allocator using `malloc()` and `free()`

```
#![feature(allocator_api)]

use std::alloc::{AllocError, Allocator, Layout};
use std::ptr::NonNull;

use libc::{free, malloc};

pub struct BasicAllocator;

unsafe impl Allocator for BasicAllocator {
    fn allocate(
        &self,
        layout: Layout,
    ) -> Result<NonNull<[u8]>, AllocError> {
        unsafe {                                                    ❶
            let ptr = malloc(layout.size() as libc::size_t);        ❷
            let slice = std::slice::from_raw_parts_mut(             ❸
                ptr as *mut u8,
                layout.size(),
            );
            Ok(NonNull::new_unchecked(slice))                       ❹
        }
    }
    unsafe fn deallocate(&self, ptr: NonNull<u8>, _layout: Layout) {
        free(ptr.as_ptr() as *mut libc::c_void);                    ❺
    }
}
```

❶ The `allocate()` method in the `Allocator` trait *does not* include the `unsafe` keyword, but we still need to make a unsafe calls. Thus, this code block is wrapped in an `unsafe {}` block.

❷ We're calling the C library's `malloc()`, and we assume normal standard alignment from the `Layout` struct.

❸ The block of memory is returned as a slice, so first we convert the raw C pointer into a Rust slice.

❹ Lastly, create and return the final pointer to the slice of bytes.

❺ `deallocate()` is essentially the reverse of `allocate()`, but this method is already marked as `unsafe` for us. The pointer must be converted from its raw Rust representation to a C pointer.

NOTE **The `Layout` struct contains `size` and `align` properties, both of which should be handled for portability. The `size` property specifies the minimum number of bytes to allocate, and the `align` property is the minimum byte alignment for a block in powers of 2. For details, refer to the Rust documentation on `Layout` at https://doc.rust-lang.org/stable/std/alloc/struct.Layout.html.**

Pay attention to the use of the `unsafe` keyword above: the `deallocate()` method includes `unsafe` as part of the function signature itself, and `allocate()` requires the use of `unsafe` within the method. In both cases, `unsafe` is required and cannot be avoided because we're

handling raw pointers and memory. `deallocate()` is marked as unsafe because if the method is called with invalid data (such as a bad pointer or incorrect layout), the behaviour is undefined and therefore considered unsafe.

In the event you need to write a custom allocator, the code above provides a starting point for you, regardless of your allocation needs.

5.8.2 Creating a custom allocator for protected memory

Let's quickly explore a more advanced example of a custom memory allocator to shed light on a scenario in which you'd want to utilize Rust's allocator API. For this example, the allocator can be applied piecewise to individual data structures, rather to the program as a whole, which allows fine-tuning for performance purposes.

In the *dryoc* crate, which I use for example purposes throughout this book, I make use of the `Allocator` trait to implement the protected memory feature of dryoc. Modern operating systems provide a number of memory protection features for developers who are writing safety or security critical systems, and in order to utilize those features in a Rust program, you would need to write your own memory allocation code. Specifically, the dryoc crate uses the *`mprotect()`* and *`mlock()`* system calls on UNIX-like systems, and the *`VirtualProtect()`* and *`VirtualLock()`* system calls on Windows. These system calls provide the ability for locking and controlling access to specific regions of memory within a process, both to code *inside* and *outside* the process. This is an important feature for code which manages sensitive data such as secret keys.

As part of the implementation of memory locking and protection features, memory must be allocated by special platform-dependent memory functions (`posix_memalign()` on UNIX, `VirtualAlloc()` on Windows), such that it's aligned to platform-specific memory pages. Additionally, in the code below, two extra memory blocks are allocated before and after the target memory region, and those blocks are locked, which provides additional protection against certain types of memory attacks. These regions can be thought of as bumpers like you would find on an automobile.

When our custom allocator is used, memory will be allocated on the heap as per the diagram shown in figure 5.2:

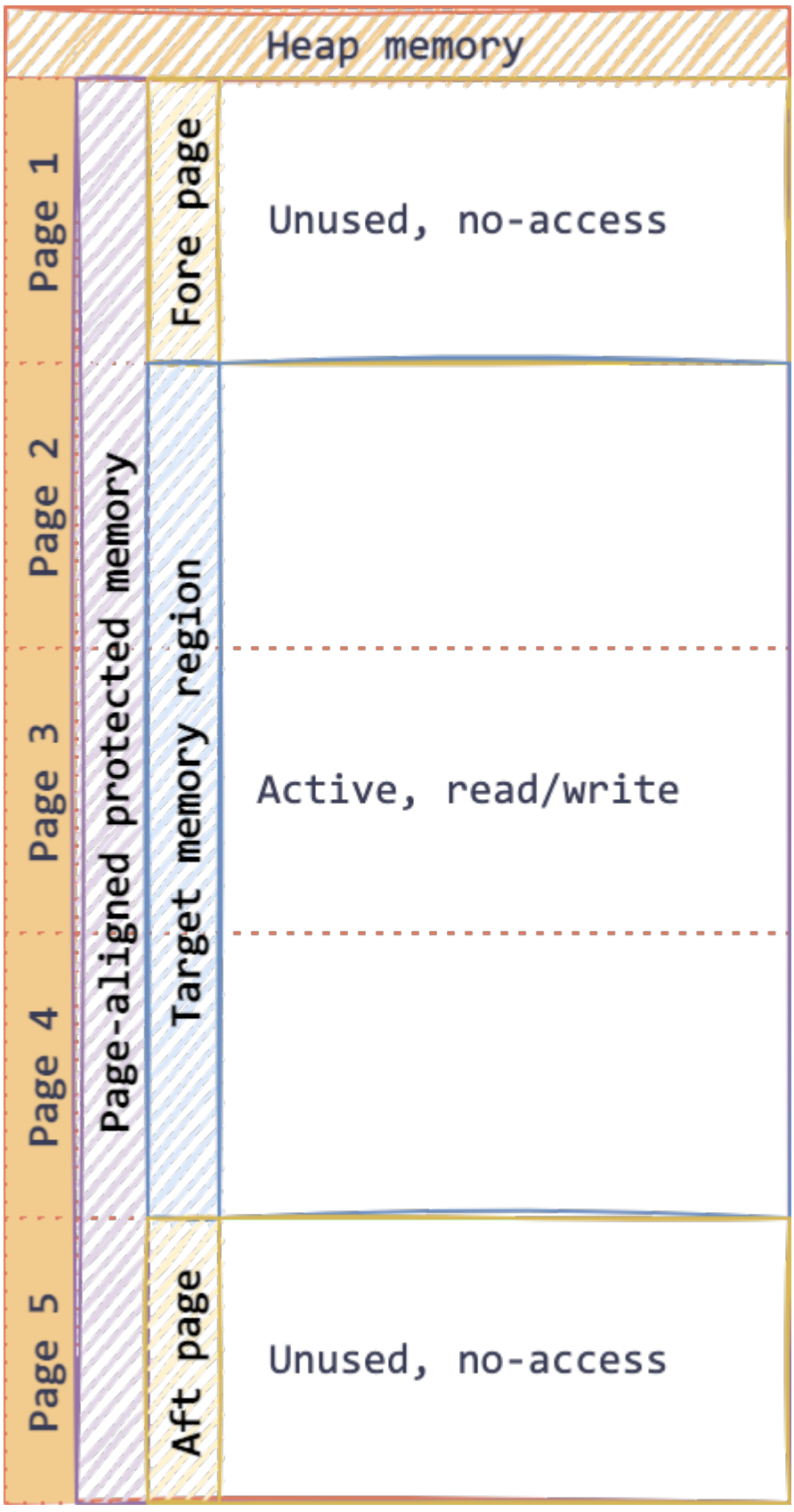

Figure 5.2 Diagram showing protected memory layout, with fore and aft regions.

The active region is a *subset* of the total memory allocated, and a subset which excludes the first

and last pages is returned as a *slice* by the allocator.

Let's examine a partial code sample of this allocator (the full code listing is included with the book's code). First, we'll examine listing 5.13:

Listing 5.13 Partial code listing for `allocate()` from page-aligned allocator, based on code from the dryoc crate

```
fn allocate(&self, layout: Layout,
) -> Result<ptr::NonNull<[u8]>, AllocError> {
    let pagesize = *PAGESIZE;
    let size = _page_round(layout.size(), pagesize) + 2 * pagesize;   ❶
    #[cfg(unix)]
    let out = {
        let mut out = ptr::null_mut();
        let ret = unsafe {
            libc::posix_memalign(&mut out, pagesize as usize, size)   ❷
        };
        if ret != 0 {
            return Err(AllocError);
        }
        out
    };
    #[cfg(windows)]
    let out = {
        use winapi::um::winnt::{MEM_COMMIT, MEM_RESERVE, PAGE_READWRITE};
        unsafe {
            winapi::um::memoryapi::VirtualAlloc(   ❸
                ptr::null_mut(), size, MEM_COMMIT | MEM_RESERVE,
                PAGE_READWRITE,
            )
        }
    };
    let fore_protected_region = unsafe {
        std::slice::from_raw_parts_mut(out as *mut u8, pagesize)
    };
    mprotect_noaccess(fore_protected_region)   ❹
        .map_err(|err| {
            eprintln!("mprotect error = {:?}, in allocator", err)
        })
        .ok();
    let aft_protected_region_offset =
        pagesize + _page_round(layout.size(), pagesize);
    let aft_protected_region = unsafe {
        std::slice::from_raw_parts_mut(
            out.add(aft_protected_region_offset) as *mut u8,
            pagesize,
        )
    };
    mprotect_noaccess(aft_protected_region)   ❺
        .map_err(|err| {
            eprintln!("mprotect error = {:?}, in allocator", err)
        })
        .ok();
    let slice = unsafe {
        std::slice::from_raw_parts_mut(
            out.add(pagesize) as *mut u8,
            layout.size(),
        )
    };
    mprotect_readwrite(slice)   ❻
        .map_err(|err| {
            eprintln!("mprotect error = {:?}, in allocator", err)
        })
        .ok();
    unsafe { Ok(ptr::NonNull::new_unchecked(slice)) }   ❼
}
```

❶ Rounds the size of the memory region to the nearest page length, adding 2 additional pages before and after the memory region.

❷ Allocates page-aligned memory on POSIX-based systems.

❸ Allocates page-aligned memory on Windows-based systems.

❹ Marks the memory page *in front* of the new region as no-access, to prevent scanning.

❺ Marks the memory page *after* the new region as no-access to prevent scanning.

❻ Marks the new region of memory as read/write.

❼ Returns the new pointer as a slice consisting of the memory location and size.

Next, let's look at the implementation for `deallocate()` in listing 5.14:

Listing 5.14 Partial code listing for `deallocate()` from page-aligned allocator, based on code from the dryoc crate

```
unsafe fn deallocate(&self, ptr: ptr::NonNull<u8>, layout: Layout) {
    let pagesize = *PAGESIZE;
    let ptr = ptr.as_ptr().offset(-(pagesize as isize));
    // unlock the fore protected region
    let fore_protected_region =
        std::slice::from_raw_parts_mut(ptr as *mut u8, pagesize);
    mprotect_readwrite(fore_protected_region)  ❶
        .map_err(|err| eprintln!("mprotect error = {:?}", err))
        .ok();
    // unlock the aft protected region
    let aft_protected_region_offset =
        pagesize + _page_round(layout.size(), pagesize);
    let aft_protected_region = std::slice::from_raw_parts_mut(
        ptr.add(aft_protected_region_offset) as *mut u8,
        pagesize,
    );
    mprotect_readwrite(aft_protected_region)  ❷
        .map_err(|err| eprintln!("mprotect error = {:?}", err))
        .ok();
    #[cfg(unix)]
    {
        libc::free(ptr as *mut libc::c_void);  ❸
    }
    #[cfg(windows)]
    {
        use winapi::shared::minwindef::LPVOID;
        use winapi::um::memoryapi::VirtualFree;
        use winapi::um::winnt::MEM_RELEASE;
        VirtualFree(ptr as LPVOID, 0, MEM_RELEASE);  ❹
    }
}
```

❶ Returns the fore memory page to read/write, the default state.

❷ Returns the aft memory page to read/write, the default state.

❸ Releases the page-aligned memory, on POSIX-based systems.

❹ Releases the page-aligned memory on Windows-based systems.

The code sample above is based on code from the dryoc crate; you can find the full code listing on GitHub at https://github.com/brndnmtthws/dryoc/blob/main/src/protected.rs, which may

include future improvements.

SIDEBAR **Using the `cfg` and `cfg_attr` attributes, and the `cfg` macro for conditional compilation**

We've talked about attributes throughout the book, but it's a good time to take a moment to discuss `cfg` in more depth, as seen in the custom allocator example.

If you're coming from a language like C or C++, you're likely familiar with using macros to enable or disable code at compile time (such as `#ifdef FLAG { … } #endif`). Enabling and disabling features at compile time is a common pattern, especially for compiled languages that need access to OS-specific features (as in the custom allocator example). Rust's equivalent features look similar to but behave differently from what you may have seen in C and C++.

Rust provides 3 built-in tools for handling conditional code compilation:

- The `cfg` attribute, which conditionally includes the attached code (i.e., the item on the following line of code whether it be a block or statement)
- The `cfg_attr` attribute, which behaves like `cfg` except that it allows you to set new compiler attributes based on the existing ones
- The `cfg` macro which returns true or false at compile time

To illustrate its use, consider the following example:

Listing 5.15 Conditional compilation example

```
#[cfg(target_family = "unix")]
fn get_platform() -> String {
    "UNIX".into()
}

#[cfg(target_family = "windows")]
fn get_platform() -> String {
    "Windows".into()
}

fn main() {
    println!("This code is running on a {} family OS", get_platform());
    if cfg!(target_feature = "avx2") {
        println!("avx2 is enabled");
    } else {
        println!("avx2 is not enabled");
    }
    if cfg!(not(any(target_arch = "x86", target_arch = "x86_64"))) {
        println!("This code is running on a non-Intel CPU");
    }
}
```

In the example above, the `cfg` attribute applies to the entire function block for `get_platform()`, hence it appearing twice. We use the `cfg` macro to test if the `avx2` target feature is enabled and if we're using a non-Intel architecture.

Shorthand configuration predicates are defined by the compiler, such as `unix` and `windows`, as shown in the custom allocator example. In other words, rather than writing `#[cfg(target_family = "unix")]`, you can use `#[cfg(unix)]`. A full list of the configuration values for your target CPU can be obtained by running `rustc --print=cfg -C target-cpu=native`.

Predicates may also be combined using `all()`, `any()`, and `not()`. `all()` and `any()` accept a list of predicates, whereas `not()` accepts 1 predicate. For example, you can use `#[cfg(not(any(target_arch = "x86", target_arch = "x86_64")`

The full listing of the compile-time configuration options can be found at https://doc.rust-lang.org/reference/conditional-compilation.html.

5.9 Summary

- `Box` and `Vec` provide methods to allocate memory on the heap; `Vec` should be preferred when you need a list of items, otherwise use `Box` for a singular item
- The `Clone` trait can be used to provide deep copying of data structures in Rust
- `Rc` and `Arc` provide reference counted smart pointers for shared ownership
- `Cell` and `RefCell` provide an escape hatch for the interior mutability problem when you need to mutate data inside an immutable structure, but only for single-threaded applications
- `Mutex` and `RwLock` provide synchronization primitives which can be used with `Arc` to enable internal mutability
- The `Allocator` and `GlobalAlloc` APIs provide a way to customize memory allocation behaviour in Rust

In the following table, I have summarized the core smart pointer and memory container types to guide you when you are deciding which to use. You can refer to it as you learn more about Rust and start experiment with more advanced memory management.

Table 5.2 Summarizing Rust's smart pointers and containers

Type	Kind	Description	When to use?	Single or multi-threaded?
`Box`	Pointer	Heap-allocated smart pointer	Any time you need to store a single object on the heap (and not in a container such as a `Vec`)	single
`Cow`	Pointer	Smart pointer with clone-on-write, which can be used with owned or borrowed data	When you need heap-allocated data with clone-on-write functionality	single
`Rc`	Pointer	Reference-counted heap allocated smart pointer that enables shared ownership	When you need shared-ownership of heap-allocated data	single
`Arc`	Pointer	Atomically reference-counted heap allocated smart pointer that enables shared ownership	When you need shared-ownership of heap-allocated data across threads	multi
`Cell`	Container	Memory container that enables interior mutability using move	When you need to enable interior mutability of data within a smart pointer using move	single
`RefCell`	Container	Memory container that enables interior mutability using references	When you need to enable interior mutability using references	single
`Mutex`	Container	Mutual exclusion primitive that also enables interior mutability with a reference	When you need to synchronize data sharing across threads	multi
`RwLock`	Container	Mutual exclusion primitive that provides distinction between readers and writers, and enables interior mutability with a reference	When you need reader/writer locking across threads	multi

Before you move on to the next chapters, take some time to understand the linked list examples in this chapter. I suggest trying to implement them on your own, and only referring to the sample code as needed. Once you get comfortable with memory management in Rust, you'll find everything else becomes much easier.

6 Unit testing

This chapter covers

- Understanding how unit testing in Rust differs from other languages
- Reviewing Rust's unit testing features
- Exploring testing frameworks
- Dos and don'ts for unit testing in Rust
- Unit testing with parallel code
- Writing unit tests with refactoring in mind
- Exploring tools to help with refactoring
- Measuring code covered by tests
- Testing strategies for dealing with Rust's rapidly changing ecosystem

Unit testing is one way to improve code quality by catching regressions and ensuring code meets requirements before shipping. Rust includes a built-in unit testing framework to make your life easier. In this chapter we'll review some of the features Rust provides, and discuss some of the shortfalls of Rust's unit testing framework and how to overcome them.

6.1 How testing is different in Rust

Before we jump into the details of Rust's unit testing features, we should talk about the differences between Rust and other languages and how it relates to unit testing. For those coming from languages like Haskell or Scala, you may find Rust has similar properties when it comes to testing. Compared to most languages, however, Rust varies greatly in that the kind of unit tests you might normally see in other languages isn't necessary in Rust.

To elaborate on that, there are many cases in Rust where so long as the code compiles, the code must be correct. In other words, the Rust compiler can be thought of an automatic test suite that's always applied to code. This only remains true for certain cases of tests, and there are a variety of

ways to break this contract in Rust.

The two most commons ways to undo some of Rust's safety guarantees are:

- Use of the `unsafe` keyword
- Converting compile-time errors into runtime errors

For the latter, converting compile time errors into runtime errors can happen in a variety of ways, but the most common is by using `Option` or `Result` without properly dealing with both result cases. In particular, calling `unwrap()` on these types without handling the failure case. In some cases, this is the desired behaviour, but it's also a mistake people often make simply because they don't want to spend time handling errors. To avoid these problems, the simple solution is to handle all cases and avoid calling functions that panic at runtime (such as `unwrap()`). Rust *does not* provide a way to verify that code is panic-free.

In the case of Rust's standard library, functions and methods that panic on failures are generally noted as such in the documentation. As a general rule for any kind of programming, any functions which perform I/O or nondeterministic operations may fail (or panic) at any time, and those failure cases should be handled appropriately (unless, of course, the correct way to handle the failure is to panic).

NOTE **The term "panic" in Rust means to raise an error and abort the program. If you want to force a panic yourself, the `panic!()` macro can be used. Additionally, you can use the `compile_error!()` macro to induce a compile-time error.**

Oftentimes the Rust compiler can catch errors before code ships, without the help of unit tests. What the Rust compiler cannot do, however, is catch *logic* errors. For example, the Rust compiler can detect certain cases of divide-by-zero errors, but it can't tell you when you mistakenly used division instead of multiplication.

As a general rule, the best way to write software that's both easy to test and hard to be wrong is accomplished by breaking down code into small units of computation (functions) which generally satisfy the following properties:

1. Functions should be stateless when possible
2. Functions should be idempotent in cases where they must be stateful
3. Functions should be deterministic whenever possible; the result of a function should always be the same for any given set of inputs
4. Functions which might fail should return a `Result`
5. Functions which may return no value (i.e., null) should return an `Option`

Following points 4 and 5 allow you to make heavy use of the `?` operator in Rust (`?` is shorthand

for "return early with an error result" if the result is not `Ok`), which can save you from typing a lot of code. In chapter 4 we discussed using `Result` with the `From` trait, which greatly simplifies error handling in your code. For any given function you write that returns a `Result`, you only need to write the necessary `From` trait implementation for any possible errors within the function, and then you can handle those errors appropriately with the `?` operator. Keep in mind this only works in generic situations, it may not be appropriate in cases where the error handling is specific to the function in question.

YOUR TURN **If you want to panic on an unexpected result, use the `expect()` function. `expect()` takes a message as an argument explaining why the program panicked. `expect()` is a safer alternative to `unwrap()`, and can be thought of as behaving similarly to `assert()`.**

By convention, Rust unit tests are stored in the same source file as the code being tested. That is to say, for any given struct, function, or method, its corresponding unit test would generally be within the same source file. Tests are typically located near the bottom of the file. This has a nice side effect which is that it helps you keep code relatively small and separate concerns. If you try to pack too much logic into one file, it can grow quite large especially if you have complicated tests. Once you pass the 1,000 line mark, you may need to think about refactoring.

Lastly, most of this advice is not necessarily Rust specific, it applies to all programming languages. For Rust specifically, your code should always handle return values and avoid the use of `unwrap()` except when necessary.

6.2 Review of built-in testing features

Rust provides a number of basic testing features, although you may find the built-in features lacking compared to more mature testing frameworks. One big difference between Rust and other languages is that the core Rust tooling and language includes testing features without the use of additional libraries or frameworks. In many languages, testing is an after though, and requires various additional toolings and libraries bolted on to properly test code.

Features *not* provided by Rust can usually be found in crates, however you may also find that Rust makes testing much easier overall thanks to the strict language guarantees.

Table 6.1 Summary of Rust's testing features

Feature	Description
Unit testing	Rust and cargo provide unit testing directly without the use of additional libraries using the `tests` mod within source files. The `tests` mod must be marked with `#[cfg(test)]`, and test functions must be marked with the `#[test]` attribute.
Integration testing	Rust and cargo provide integration testing, which allows testing of libraries and applications from their public interfaces. Tests are typically constructed as their own individual applications, separate from the main source code.
Documentation testing	Code samples in source code documentation using `rustdoc` are treated as unit tests, which cleverly improve the overall quality of documentation and testing simultaneously.
Cargo integration	Unit, integration, and documentation tests work with cargo automatically, and don't require additional work beyond defining the tests themselves. The `cargo test` command handles filtering, displaying assertion errors, and even parallelizes tests for you.
Assertion macros	Rust provides assertion macros such as `assert!()` and `assert_eq!()`, although these are not exclusive to tests (they can be used anywhere because they're normal Rust macros). Cargo, however, will properly handle assertion failures when running unit tests and provide helpful output messages.

Let's examine the anatomy of a simple library with a unit test to demonstrate Rust's testing features. Consider the following example, which provides a generic adder:

Listing 6.1 Code listing for a basic unit test in Rust

```
pub fn add<T: std::ops::Add<Output = T>>(a: T, b: T) -> T {   ❶
    a + b   ❶
}   ❶

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

    #[test]   ❹
    fn test_add() {
        assert_eq!(add(2, 2), 4);
    }
}
```

❶ An addition function that takes 2 parameters of the same type, and returns the result of the same type. The type `T` needs to have the `std::ops::Add` trait implemented for the same output type.

❷ Our mod `tests` contains our tests, and the `#[cfg(test)]` attribute tells the compiler this is our unit test mod.

❸ A convenient shorthand to include everything from the outer scope of this mod. You'll often see this used in tests.

❹ The `#[test]` attribute tells the compiler this function is a unit test.

The test above passes, which is great. If you run `cargo test`, the output looks like this:

Listing 6.2 Successful test run

```
$ cargo test
   Compiling unit-tests v0.1.0
   (/Users/brenden/dev/code-like-a-pro-in-rust/code/c6/6.2/unit-tests)
    Finished test [unoptimized + debuginfo] target(s) in 0.95s
     Running unittests (target/debug/deps/unit_tests-c06c761997d04f8f)

running 1 test
test tests::test_add ... ok

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

   Doc-tests unit-tests

running 0 tests

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

6.3 Testing frameworks

Rust's unit testing doesn't include the helpers, fixtures, harnesses, or parameterized testing features like you may find in other unit testing frameworks. For those, you'll either need to code your own, or try some libraries.

For basic parameterized testing, the parameterized[20] crate provides a nice interface to create tests. The test-case[21] crate provides another implementation of parameterized testing that's simple, concise, and easy to use. For fixtures, you can try the rstest[22] crate. The assert2[23] crate provides assertions inspired by the popular C++ Catch2 library.

One library worth mentioning in detail is the proptest[24] crate, which provides a Rust implementation of QuickCheck[25], a Haskell library originally released in 1999, and one which you may have encounted before. proptest isn't a 1:1 port of QuickCheck to Rust, but rather it provides equivalent functionality with some Rust-specific differences which are documented at https://altsysrq.github.io/proptest-book/proptest/vs-quickcheck.html.

Property testing can save you a lot of time by generating random test data, verifying results, and reporting back with the minimum test case required to create an error. This is a huge time saver, although it's not necessarily a replacement for testing well-known values (for example, when verifying spec compliance).

NOTE **There's no free lunch with property testing, and it comes with the trade-off of possibly having to spend more CPU cycles testing random values, as opposed to hand-picked or well-known values. You can tune the number of random values to test, but for data with a large set of possible values, it's often not practical to test every outcome.**

Let's revisit our adder example from previous section, but this time we'll try it with proptest which will provide the test data for our test function:

Listing 6.3 Code listing of adder with proptest

```
pub fn add<T: std::ops::Add<Output = T>>(a: T, b: T) -> T {
    a + b
}

#[cfg(test)]
mod tests {
    use super::*;
    use proptest::prelude::*;   ❶
    proptest! {
        #[test]
        fn test_add(a: i64, b: i64) {   ❷
            assert_eq!(add(a, b), a + b);   ❸
        }
    }
}
```

❶ Here we include the proptest library, which includes the `proptest!` macro.

❷ Our test function's parameters, `a` and `b`, will be provided by the `proptest!` macro.

❸ We assert that our adder does indeed return the result of `a + b`.

Now let's run this test again with proptest:

```
cargo test
   Compiling proptest v0.1.0
   (/Users/brenden/dev/code-like-a-pro-in-rust/code/c6/6.2/proptest)
    Finished test [unoptimized + debuginfo] target(s) in 0.59s
     Running unittests (target/debug/deps/proptest-db846addc2c2f40d)

running 1 test
test tests::test_add ... FAILED

failures:

----- tests::test_add stdout -----
# ... snip ...
thread 'tests::test_add' panicked at 'Test failed: attempt to add with
overflow; minimal failing input: a = -2452998745726535882,
b = -6770373291128239927
    successes: 1
    local rejects: 0
    global rejects: 0
', src/lib.rs:9:5

failures:
    tests::test_add

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

error: test failed, to rerun pass '--lib'
```

Uh oh, looks like our adder wasn't so great after all. Turns out it can blow up under certain circumstances (in this case, the addition operation overflowed because we're adding two signed integers of finite length). We weren't expecting this kind of failure and probably wouldn't have

caught it unless we were manually generating random data for `a` and `b`.

SIDEBAR **Arithmetic overflow in Rust**

Arithmetic operations in Rust may trip people up at first, especially when testing. There's a simple reason for this: in Rust, code compiled in debug mode (such as tests) use checked arithmetic by default. When the same code is compiled in release mode, the same code will use unchecked arithmetic. Thus, you can have code which fails when run in debug mode, but works fine (i.e., does not produce an error or crash the program) in production.

Rust's approach can be a bit confusing because of how the behaviour is different depending on how code is compiled. The rationale in Rust is that test code should be stricter to catch more bugs, but for compatibility the code should behave the way most other programs behave at runtime.

Developers sometimes take arithmetic overflow for granted, because most languages either emulate the behaviour of languages like C (which is usually referred to as wrapped arithmetic, i.e., when the integer overflows it just wraps around).

Rust provides a number of alternative arithmetic functions for primitive types, which are documented in the standard library for each type. For example, i32 provides `checked_add()`, `unchecked_add()`, `carrying_add()`, `wrapping_add()`, `overflowing_add()`, and `saturating_add()`.

To emulate the C behaviour, you can use the `Wrapping` struct (documented at https://doc.rust-lang.org/std/num/struct.Wrapping.html) or by calling the corresponding method for each type and operation.

This behaviour is document in RFC 560 at https://github.com/rust-lang/rfcs/blob/master/text/0560-integer-overflow.md .

We have a few options for fixing the test above, but the easiest one is to just explicitly wrap overflows (i.e., follow the C behaviour of integer overflow). Let's update our code to look like this:

```
extern crate num_traits;  ❶
use num_traits::ops::wrapping::WrappingAdd;

pub fn add<T: WrappingAdd<Output = T>>(a: T, b: T) -> T {  ❷
    a.wrapping_add(&b)
}

#[cfg(test)]
mod tests {
    use super::*;
    use proptest::prelude::*;
    proptest! {
        #[test]
        fn test_add(a: i64, b: i64) {
            assert_eq!(add(a, b), a.wrapping_add(b));  ❸
        }
    }
}
```

❶ We're relying on the num_traits crate which provides the `WrappingAdd` trait

❷ The trait bound is switched from `Add` to `WrappingAdd`

❸ The test needs to be updated so it also using `wrapping_add()`

For the code above, we've added the num_traits crate which is a small library that provides us with the `WrappingAdd` trait. The Rust standard library doesn't have an equivalent trait available, and it's difficult to create a generic function this way without one (we'll explore traits in greater depth in chapters 8 & 9).

If we run our code now, it passes as expected:

```
cargo test
   Compiling wrapping-adder v0.1.0
   (/Users/brenden/dev/code-like-a-pro-in-rust/code/c6/6.2/wrapping-adder)
    Finished test [unoptimized + debuginfo] target(s) in 0.65s
     Running unittests (target/debug/deps/wrapping_adder-5330c09f59045f6a)

running 1 test
test tests::test_add ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out;
finished in 0.01s

     Running unittests (target/debug/deps/wrapping_adder-a198d5a6a64245d9)

running 0 tests

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

   Doc-tests wrapping-adder

running 0 tests

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

6.4 What not to test, or why the compiler knows better than you

Rust is a statically typed language, which brings with it some major advantages, especially when it comes to testing. One way to think about statically typed languages versus dynamically typed is that the compiler does the work of limiting the set of possible inputs and outputs to any given statement or block of code by analyzing the source code before it runs. The possible set of inputs and outputs are constrained by type specifications. That is to say, a string can't be an integer and vice versa. The compiler verifies that the types match what's expected, and references are valid. You don't need to worry about mixing up strings and integers at runtime because the compiler doesn't let it happen; this frees you (the developer) from having to worry about a host of problems, provided you use types correctly.

In dynamically typed languages, type errors are one of the most common problems. In interpreted languages, the combination of invalid syntax and type errors create a potent opportunity for runtime errors that can be difficult to catch before code is shipped. Many dynamic languages have been retrofitted with static analyzer tools, but they're not always adequately strict or thorough to catch common errors.

When it comes to testing, we never need to test anything the compiler or borrow checker tests for us. For example, we don't need to check if an integer is an integer, or a string is a string. We don't need to check that references are valid, or if data is being mutated by two different threads (a race condition).

Let's be clear that this doesn't mean you don't need to test, it just means most of what you test in Rust is *logic*, rather than type validation or memory use. It's true that you still need to perform type conversions which might fail, but handling these is a matter of logic. Rust's ergonomics make it hard to handle things that might fail improperly.

Testing properly in Rust begins with effectively using Rust's type system. Overuse of `Option`, `unwrap()`, or unsafe code can lead to harder to find bugs, especially if you use these features as a way to avoid handling edge cases. Stateful operations and I/O need to be checked and handled appropriately (as a good habit, functions or methods performing I/O should probably return a `Result`).

6.5 Handling parallel test special cases and global state

When Cargo runs unit tests, it does so in parallel. Rust uses threads to execute multiple tests simultaneously in order to speed up testing. Most of the time this works transparently, and you don't need to worry about it. However, we sometimes find ourselves needing to create global state, or fixtures for tests, and doing so may required shared state.

Rust provides a couple of facilities for handling this problem: one is to create your own `main()`

function for testing (by effectively override Rust's *libtest*, which is the built-in testing library in Rust, and not something you typically interact with directly). That option, however, is probably more trouble than it's worth, so instead I'll direct you to a different alternative: the handy lazy_static[26] crate.

YOUR TURN **If you do want to provide your own `main()` for testing, it can be done so by disabling Rust's built in harness libtest with `harness = false` in the target settings. Some details on how to do this can be found in the rustc documentation at https://doc.rust-lang.org/rustc/tests/index.html and libtest at https://doc.rust-lang.org/test/index.html.**

If, by some chance, you haven't already encounted lazy_static at this point, then you'll be pleased to learn about it now. The lazy_static crate makes the creation of static variables in Rust much easier. Creating global shared state in Rust is somewhat tricky because you sometimes need to initialize static structures at runtime. To accomplish this, you can create a static reference and update that reference when it's first accessed, which is what lazy_static does.

To illustrate the problem with global state, consider the following code:

Listing 6.4 Code listing of unit test with global count

```
#[cfg(test)]
mod tests {
    static mut COUNT: i32 = 0;   ❶

    #[test]
    fn test_count() {
        COUNT += 1;   ❷
    }
}
```

❶ Defines a static mutable counter variable

❷ Increments the counter within our test

The code above fails to compile with the following error:

```
error[E0133]: use of mutable static is unsafe and requires unsafe function
or block
 --> src/lib.rs:7:9
  |
7 |         COUNT += 1;
  |         ^^^^^^^^^^^ use of mutable static
  |
  = note: mutable statics can be mutated by multiple threads: aliasing
  violations or data races will cause undefined behavior
```

The compiler is correctly catching the error here. If you wrote equivalent code in C, it would compile and run without complaint (and probably work most of the time…until it doesn't).

We've got a couple options to fix the code. In this case, we've just got a simple count, so we can simply use an atomic integer instead (which is thread safe). Seems easy enough, right? Let's try the following:

```
#[cfg(test)]
mod tests {
    use std::sync::atomic::{AtomicI32, Ordering};
    static mut COUNT: AtomicI32 = AtomicI32::new(0);   ❶

    #[test]
    fn test_count() {
        COUNT.fetch_add(1, Ordering::SeqCst);   ❷
    }
}
```

❶ Uses an Atomic integer, provided by Rust's standard library

❷ Performs a fetch and add operation, which increments the atomic integer. `Ordering::SeqCst` tells the compiler how to synchronize operations, documented at https://doc.rust-lang.org/std/sync/atomic/enum.Ordering.html.

If we try compiling our updated test, it prints the exact same error (`use of mutable static`). So what gives? rustc is being *very* strict: it's complaining about the *ownership* of the `COUNT` variable, which itself doesn't implement the `Send` trait. We'll have to introduce `Arc` to implement `Send` as well.

SIDEBAR **Rust's Send and Sync traits**

Rust provides 2 important traits for handling shared state across threads: *Send* and *Sync*. These traits are what the compiler uses to provide Rust's thread safety guarantees, and you'll need to understand them when working with multithreaded Rust code and shared state.

These traits are defined as such:

- `Send` **marks objects that can be safely moved between threads**
- `Sync` **marks objects that can be safely shared between threads**

For example, if you want to move a variable from one thread to another, it needs to be wrapped in something that implements `Send`. If you want to have shared references to the same variable across threads, you'll need to wrap it in something that implements `Sync`.

These traits are automatically derived by the compiler where appropriate. You don't need to implement them directly, but rather you can use combinations of `Arc`, `Mutex`, and `RwLock` (discussed in chapter 5) to achieve thread safety.

Let's update our code again, now that we've realized we need to use `Arc`:

```
#[cfg(test)]
mod tests {
    use std::sync::atomic::{AtomicI32, Ordering};
    use std::sync::Arc;

    static COUNT: Arc<AtomicI32> = Arc::new(AtomicI32::new(0));

    #[test]
    fn test_count() {
        let count = Arc::clone(&COUNT);   ❶
        count.fetch_add(1, Ordering::SeqCst);
    }
}
```

❶ We need to clone the `Arc` before we can use it, to obtain a reference in this thread context

If we try to compile this we're going to be disappointed again with a new error:

```
error[E0015]: calls in statics are limited to constant functions, tuple
structs and tuple variants
 --> src/lib.rs:6:38
  |
6 |     static COUNT: Arc<AtomicI32> = Arc::new(AtomicI32::new(0));
  |                                     ^^^^^^^^^^^^^^^^^^^^^^^^^^
```

If you reached this point on your own, you may feel like giving up and quitting. However, the solution is pretty simple: lazy_static. The compiler doesn't let us create globals that aren't constants, so we need to either write custom code to do the initialization at runtime, or we can just use lazy_static. Let's update the test one more time:

```
#[cfg(test)]
mod tests {
    use lazy_static::lazy_static;
    use std::sync::atomic::{AtomicI32, Ordering};
    use std::sync::Arc;
    lazy_static! {   ❶
        static ref COUNT: Arc<AtomicI32> = Arc::new(AtomicI32::new(0));   ❷
    }

    #[test]
    fn test_count() {
        let count = Arc::clone(&COUNT);
        count.fetch_add(1, Ordering::SeqCst);
    }
}
```

❶ The `lazy_static!` macro is used to wrap our static variable definitions

❷ When initializing with `lazy_static!`, you supply a code block returning the initialized object. In this case it all fits on one line, so the braces (`{ … }`) are omitted.

Presto! Now our code compiles and runs *safely*. lazy_static takes care of the details of initializing the data at runtime: when the variable is first accessed, it's initialized automatically and we can use it globally. To understand what lazy_static does, let's view the code generated by the macro

with `cargo expand` (introduced in chapter 3):

```
#[allow(missing_copy_implementations)]
#[allow(non_camel_case_types)]
#[allow(dead_code)]
struct COUNT {
    __private_field: (),
}
#[doc(hidden)]
static COUNT: COUNT = COUNT {
    __private_field: (),
};
impl ::lazy_static::__Deref for COUNT {   ❶
    type Target = Arc<AtomicI32>;
    fn deref(&self) -> &Arc<AtomicI32> {
        #[inline(always)]
        fn __static_ref_initialize() -> Arc<AtomicI32> {
            Arc::new(AtomicI32::new(0))   ❷
        }
        #[inline(always)]
        fn __stability() -> &'static Arc<AtomicI32> {
            static LAZY: ::lazy_static::lazy::Lazy<Arc<AtomicI32>> =
                ::lazy_static::lazy::Lazy::INIT;   ❸
            LAZY.get(__static_ref_initialize)   ❸
        }
        __stability()
    }
}
impl ::lazy_static::LazyStatic for COUNT {
    fn initialize(lazy: &Self) {
        let _ = &**lazy;
    }
}
```

❶ lazy_static implements the `Deref` trait (within its code `__Deref` is aliased to the core library `Deref`)

❷ This block will be replaced with the initialization code supplied by us (in our case, a single line)

❸ lazy_static uses the `std::sync::Once` primitive internally, from the Rust core library, which is initialized at this point

Examining lazy_static's source code, we can see it's based on the `std::sync::Once` primitive (provided by the standard library). We can drop the superfluous `Arc` that we added in the previous step, because lazy_static provides `Send`. The final result when using lazy_static looks like this:

```
#[cfg(test)]
mod tests {
    use lazy_static::lazy_static;
    use std::sync::atomic::{AtomicI32, Ordering};
    lazy_static! {
        static ref COUNT: AtomicI32 = AtomicI32::new(0);
    }

    #[test]
    fn test_count() {
        COUNT.fetch_add(1, Ordering::SeqCst);
    }
}
```

And while lazy_static helps us solve the problem of sharing global state, it doesn't help us with synchronizing the tests themselves. For example, if you want to ensure your tests execute one at a time, you'll have to either implement your own `main()` to run your tests, instruct libtest to run the tests with only 1 thread, or you can synchronize your tests using a mutex:

```
#[cfg(test)]
mod tests {
    use lazy_static::lazy_static;
    use std::sync::Mutex;
    lazy_static! {
        static ref MUTEX: Mutex<i32> = Mutex::new(0);
    }
    #[test]
    fn first_test() {
        let _guard = MUTEX.lock().expect("couldn't acquire lock");
        println!("first test is running");
    }
    #[test]
    fn second_test() {
        let _guard = MUTEX.lock().expect("couldn't acquire lock");
        println!("second test is running");
    }
}
```

If you run the code above repeatedly with `cargo test—--nocapture`, you may notice that the output doesn't always print in the same order. That's because we can't guarantee the order of execution (libtest is still trying to run these tests in parallel). If you need tests to run in a particular order, you need to either use a barrier or condition variable, or implement your own `main()` function to run your tests.

As a final note, let it be said that unit tests *shouldn't* require synchronization or shared state. If you find yourself doing this, you may need to consider whether your design needs to be refactored.

6.6 Thinking about refactoring

One value proposition of unit testing is catching regressions–code changes that break existing features or behaviours–*before* software ships. Indeed, if your unit tests cover the software's specification in its entirety, any change to that software which does not conform to the specification will result in test failures.

Code refactoring–which I'll define here as code changes which don't affect the behaviour of public interfaces to the software–is common practice and carries with it advantages as well as risks. The main risk of refactoring is the introduction of regressions. The benefits of refactoring can be some combination of improved code quality, faster compilation, and better performance.

We can employ various strategies when writing tests to improve the quality of our software. One strategy is to test public interfaces in addition to private or internal interfaces. This works especially well if you can achieve near 100% coverage (and we'll discuss code coverage later in

this chapter).

In practical software development, unit tests break frequently and can often consume a great deal of development time to debug, fix, and maintain. For that reason, only testing what *needs* to be tested can–perhaps counterintuitively–save you time and provide an equivalent or better level of software quality. Figuring out what needs to be tested can be accomplished by analyzing test coverage, determining what's required by the specifications, and removing anything that doesn't need to be there (provided it's not a breaking change).

With good testing, we can refactor mercilessly with confidence. Too much testing makes our software inflexible and we spin our wheels managing tests. Combining automated testing tools such as property based testing, fuzz testing (discussed in the next chapter), and code coverage analysis gives us a great deal of quality and flexibility without requiring superpowers.

6.7 Refactoring tools

So now you have some wonderful tests, a clean API, and you want to start improving the internals of your software by cleaning things up. Some refactorings are harder than others, but there are a few tools we can use to make things smoother.

Before we discuss *which* tools to use, we need to break down the process of refactoring into types of refactorings. Some examples of common refactoring tasks are:

- Reformating: adjusting whitespace and rearranging symbols for readability
- Renaming: changing the names of variables, symbols, constants
- Relocating: moving code from one location to another within the source tree, possibly into different crates
- Rewriting: the process of completely rewriting sections of code or algorithms

6.7.1 Reformating

For code formatting, the preferred tool is rustfmt (introduced in chapter 3). You will rarely need to manually reformat Rust code. rustfmt can be configured according to your preferences; review the rustfmt section in chapter 3 for details on how to adjust the rustfmt settings to your preferences.

Using rustfmt is as simple as running `cargo fmt` as needed, or it can be integrated directly into your editor or IDE using `rust-analyzer`.

6.7.2 Renaming

Let's discuss renaming, can sometimes be tricky in complex situations. Most code editors include some type of find/replace tool to apply code changes (or you can do this from the command line using `sed` or some other command), but these aren't always the best way to do big refactorings. Regular expressions are very powerful, but sometimes we need something more contextually aware.

The `rust-analyzer` tool (first introduced in chapter 3) can intelligently rename symbols, and it also provides a *structural search and replace* tool (documented at https://rust-analyzer.github.io/manual.html#structural-search-and-replace). You can use both of these directly from your IDE or code editor. In VS Code, renaming a symbol by selecting it with the cursor and pressing F2 or using the context menu to select "Rename Symbol".

Using the structural search and replace feature of `rust-analyzer` can be accomplished either through the command palette, or by adding a comment with the replacement string. The replacement is applied to the entire workspace by default, which makes refactoring a snap. `rust-analyzer` will parse the syntax tree to find matches and perform replacements on expressions, types, paths, or items, in a way that doesn't introduce syntax errors. A substitution is only applied if the result is valid.

For example, using the Mutex guard example from earlier in this chapter we can use the `$m.lock() => Mutex::lock(&$m)` substitution as shown in figure 6.1 below:

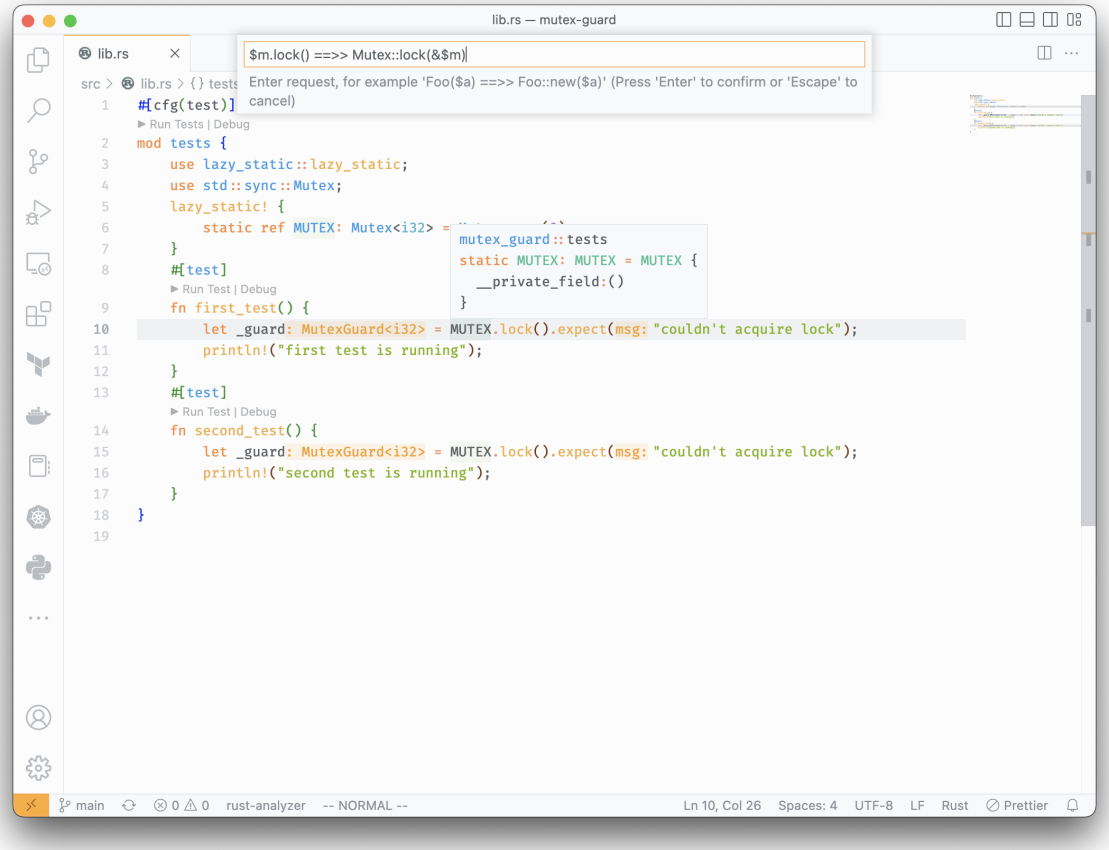

Figure 6.1 Structural substitution with `rust-analyzer`, before applying

After applying the substitution, we get the result shown in figure 6.2:

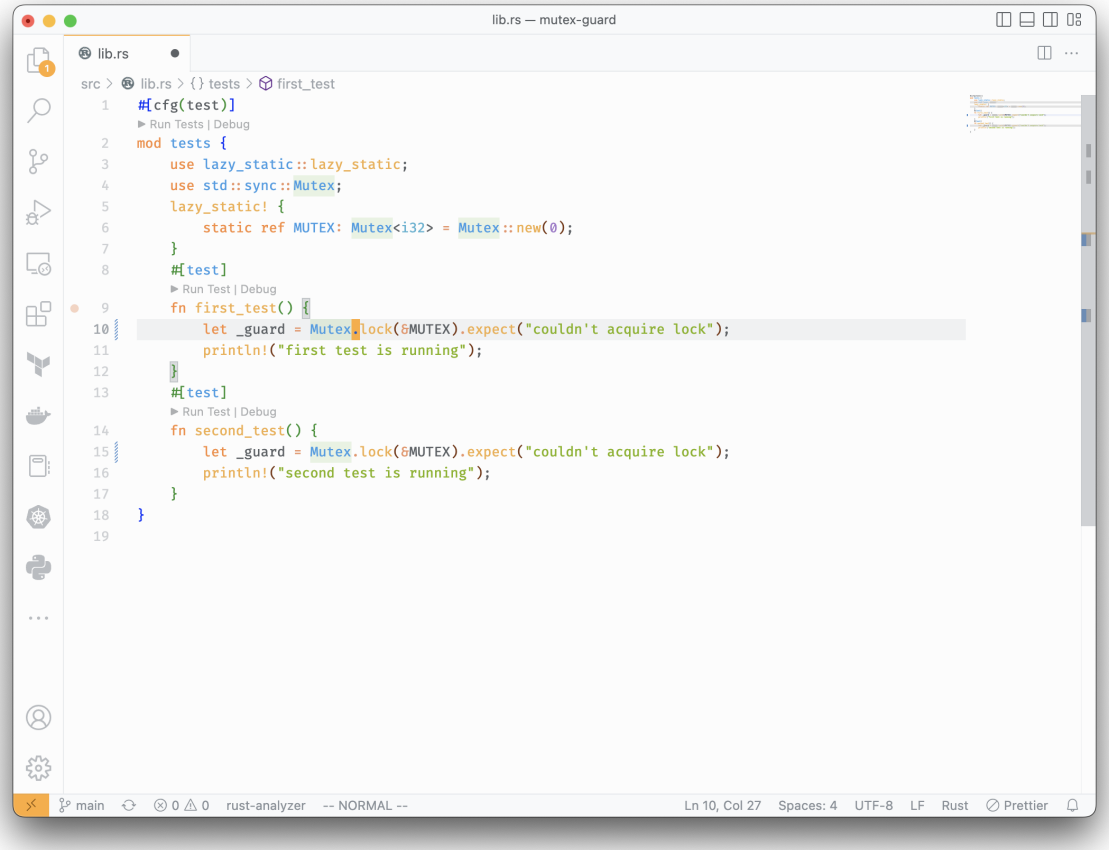

Figure 6.2 Structural substitution with `rust-analyzer`, after applying

In the example above, calling `MUTEX.lock()` and `Mutex::lock(&MUTEX)` are equivalent, but some might prefer the latter form. The structural search and replace is contextual, as you can see in the example above where I only specify `Mutex::lock()` instead of `std::sync::Mutex::lock()`, and `rust-analyzer` knows I'm asking for `std::sync::Mutex::lock()` because of the `use std::sync::Mutex` statement on line 4.

6.7.3 Relocating

At the time of writing, `rust-analyzer` doesn't have any features for relocating or moving code. For example, if you want to move a struct and its methods to a different file or module, you'll need to do this process manually.

I wouldn't normally recommend non-community projects, but I feel it's worth mentioning that the IntelliJ IDE Rust plugin, however, *does* provide a move feature for relocating code, documented at https://plugins.jetbrains.com/plugin/8182-rust/docs/rust-refactorings.html#move-refactoring. This plugin is specific to IntelliJ and (to the best of my knowledge) can't be used with other editors.

6.7.4 Rewriting

If you find yourself needing to rewrite large swaths of code, or individual algorithms, a great way to test the new code works just like the old code is to use the proptest crate, which we discussed earlier in this chapter. Consider the following implementation of the FizzBuzz[27] algorithm and corresponding test:

Listing 6.5 FizzBuzz with unit test

```
fn fizzbuzz(n: i32) -> Vec<String> {
    let mut result = Vec::new();

    for i in 1..(n + 1) {
        if i % 3 == 0 && i % 5 == 0 {
            result.push("FizzBuzz".into());
        } else if i % 3 == 0 {
            result.push("Fizz".into());
        } else if i % 5 == 0 {
            result.push("Buzz".into());
        } else {
            result.push(i.to_string());
        }
    }

    result
}

#[cfg(test)]
mod tests {
    use super::*;
    #[test]
    fn test_fizzbuzz() {
        assert_eq!(fizzbuzz(3), vec!["1", "2", "Fizz"]);
        assert_eq!(fizzbuzz(5), vec!["1", "2", "Fizz", "4", "Buzz"]);
        assert_eq!(
            fizzbuzz(15),
            vec![
                "1", "2", "Fizz", "4", "Buzz", "Fizz", "7", "8", "Fizz",
                "Buzz", "11", "Fizz",
                "13", "14", "FizzBuzz"
            ]
        )
    }
}
```

We're pretty confident the algorithm above works, but we want to write a different version of the code. So we write our new implementation like so, using a `HashMap` (with the same unit test):

```
fn better_fizzbuzz(n: i32) -> Vec<String> {
    use std::collections::HashMap;
    let mappings = HashMap::from([(3, "Fizz"), (5, "Buzz")]);
    let mut result = vec![String::new(); n as usize];
    let mut keys: Vec<&i32> = mappings.keys().collect();
    keys.sort();
    for i in 0..n {
        for key in keys.iter() {
            if (i + 1) % *key == 0 {
                result[i as usize].push_str(mappings.get(key)
                  .expect("couldn't fetch mapping"));
            }
        }
        if result[i as usize].is_empty() {
            result[i as usize] = (i + 1).to_string();
        }
    }

    result
}
```

Our new implementation is a little more complicated, and while it passes all the test cases, we aren't as confident that it works. Here's where propest comes in: we can just generate test cases using proptest, and compare that to the original implementation:

```
use proptest::prelude::*;
proptest! {
    #[test]
    fn test_better_fizzbuzz_proptest(n in 1i32..10000) {   ❶
        assert_eq!(fizzbuzz(n), better_fizzbuzz(n))   ❷

    }
}
```

❶ We limit the range of values from 1 to 10,000 for this test so it doesn't run for too long.

❷ Here we just compare the result of our old and new algorithm, which we expect to always be the same.

6.8 Code coverage

Code coverage analysis is an important tool in assessing the quality and effectiveness of your tests and code. We can automatically generate code coverage reports using a crate called Tarpaulin[28], which is provided as a Cargo command. Once installed with `cargo install cargo-tarpaulin`, you can start generating coverage reports.

Using the code from the previous section, we can generate a local HTML coverage report using `cargo tarpaulin --out Html`, the result of which which looks like this:

/Users/brenden/dev/code-like-a-pro-in-rust-book/c06/rewriting-fizzbuzz/src	Covered: 24 of 24 (100.00%)

Path	Coverage
lib.rs	24 / 24 (100.00%)

Figure 6.3 Summary of coverage report

Back /Users/brenden/dev/code-like-a-pro-in-rust-book/c06/rewriting-fizzbuzz/src/lib.rs	Covered: 24 of 24 (100.00%)

```
fn fizzbuzz(n: i32) -> Vec<String> {
    let mut result = Vec::new();

    for i in 1..(n + 1) {
        if i % 3 == 0 && i % 5 == 0 {
            result.push("FizzBuzz".into());
        } else if i % 3 == 0 {
            result.push("Fizz".into());
        } else if i % 5 == 0 {
            result.push("Buzz".into());
        } else {
            result.push(i.to_string());
        }
    }

    result
}

fn better_fizzbuzz(n: i32) -> Vec<String> {
    use std::collections::HashMap;
    let mappings = HashMap::from([(3, "Fizz"), (5, "Buzz")]);
    let mut result = vec![String::new(); n as usize];
    let mut keys: Vec<&i32> = mappings.keys().collect();
    keys.sort();
    for i in 0..n {
        for key in keys.iter() {
            if (i + 1) % *key == 0 {
                result[i as usize].push_str(
                    mappings.get(key).expect("couldn't fetch mapping"),
                );
            }
        }
        if result[i as usize].is_empty() {
            result[i as usize] = (i + 1).to_string();
```

Figure 6.4 Coverage report for lib.rs detail

Our report shows 100% coverage for lib.rs, which means every line of code has been tested by our unit tests.

These reports can either be examine locally, or integrated with a CI/CD system to track code coverage over time. Services like Codecov[29] and Coveralls[30] offer a free tier for open source

projects. The dryoc crate, for example, uses Codecov, which can be viewed at https://app.codecov.io/gh/brndnmtthws/dryoc/. These services track coverage changes over time, integrate with GitHub pull requests, and make it easy to measure progress.

One final note on code coverage: achieving 100% coverage shouldn't be your end goal. In fact, sometimes it can be nearly impossible to test every line of code. Coverage data can be used to see if you're improving over time, or at least not getting worse, but the number itself is an arbitrary metric that doesn't have qualitative substance. As Voltaire said: perfect is the enemy of good.

6.9 Dealing with a changing ecosystem

Rust is continuously improving and being updated, both in terms of the language itself and its core libraries, as well as all the crates available in the Rust ecosystem. While it's great to be on the cutting edge, this brings with it some challenges. In particular, maintaining backward and forward compatibility can be tricky.

Unit testing plays an important role in continuous maintenance, especially when dealing with moving targets. You may be tempted to simply pin dependency versions and avoid updates, but this will do more harm than good in the long run, especially as dependencies can intertwine.

Even a small number of tests go a long way in helping detect regressions, especially from third party library updates, or even language changes that you weren't expecting.

6.10 Summary

- Rust's strong static typing, strict compiler, and borrow checker lessen the burden of unit testing: runtime type errors don't need to be tested like they do in other languages
- The built-in testing features are minimal, but several crates exist to augment and automate unit testing
- Rust's libtest runs unit tests in parallel, which provides a nice speedup for normal situations, but code that's sensitive to timing or requires synchronization needs be handled accordingly
- Property testing can greatly limit the amount of time and effort spent maintaining unit tests, and provide a higher level of assurance
- Measuring and analyzing code coverage over time allows you to quantify the effectiveness of unit tests
- Unit tests help ensure third party libraries and crates function as expected after upgrades

7 Integration testing

This chapter covers

- Understanding differences between unit testing and integration testing
- Using integration testing effectively
- Comparing Rust's built-in integration testing to external testing
- Exploring libraries and tools for integration testing
- Fuzzing your tests

In chapter 6 we discussed unit testing in Rust. In this chapter we'll discuss how to use integration testing in Rust, and how it compares to unit testing. Both unit testing and integration testing are powerful strategies to improve the quality of your software. They are often used together with slightly different goals.

Integration testing can sometimes be a little more difficult because it may require more work to create harnesses and test cases, depending on the type of software being tested. It's more common to find unit tests than integration tests, but Rust provides the basic tools you need to write effective integration tests without spending too much time on boilerplate and harnesses.

We'll also explore some libraries that can help turbocharge your integration testing without require much additional work.

7.1 Comparing integration and unit testing

Integration testing is the testing of individual modules or groups from their public interfaces. This is in contrast to unit testing, which is the testing of the smallest testable components within software, and may include private interfaces. Public interfaces are those which are exposed to external consumers of software, such as the public library interfaces, or the CLI commands in the case of a command line application.

In Rust, integration tests share very little with unit tests. Unlike unit tests, integration tests are located outside of the main source tree. Rust treats integration tests as a separate crate, thus they only have access to the publicly exported functions and structures.

Let's write a quick generic implementation of the quicksort algorithm[31], which many of us know and love from our computer science studies, as shown in listing 7.1

Listing 7.1 Code listing of quicksort implemented in Rust

```
pub fn quicksort<T: std::cmp::PartialOrd + Clone>(slice: &mut [T]) {   ❶
    if slice.len() < 2 {
        return;
    }
    let (left, right) = partition(slice);
    quicksort(left);
    quicksort(right);
}

fn partition<T: std::cmp::PartialOrd + Clone>(
    slice: &mut [T]
) -> (&mut [T], &mut [T]) {   ❷
    let pivot_value = slice[slice.len() - 1].clone();
    let mut pivot_index = 0;
    for i in 0..slice.len() {
        if slice[i] <= pivot_value {
            slice.swap(i, pivot_index);
            pivot_index += 1;
        }
    }
    if pivot_index < slice.len() - 1 {
        slice.swap(pivot_index, slice.len() - 1);
    }

    slice.split_at_mut(pivot_index - 1)
}
```

❶ This is our public `quicksort()` function, denoted as such by the `pub` keyword.

❷ Our private `partition()` function is not accessible outside of the local scope.

Integration tests are located within the `tests` directory at the top-level of the source tree. These tests are automatically discovered by Cargo. An example directory structure for a small library (in `src/lib.rs`) and a single integration test would look like this:

```
$ tree
.
 Cargo.lock
 Cargo.toml
 src
    lib.rs   ❶
 tests   ❷
     quicksort.rs   ❸

2 directories, 4 files
```

❶ Contains the source code for our library

❷ Integration tests are located within the `tests` directory

❸ `quicksort.rs` contains our integration tests

Test functions are marked with the `#[test]` attribute, which will be run by Cargo automatically. You can either use the automatically provided `main()` function from libtest, or supply your own as with unit tests. Cargo handles these integration tests as separate crates. You can have multiple separate sets of crates by creating directories within `tests`, each containing their own separate integration tests.

Like with unit tests, we typically use the assertion macros (`assert!()` and `assert_eq!()`) to verify results. Here's how the integration test looks in practice in listing 7.2:

Listing 7.2 Code sample of integration test for quicksort implementation

```
use quicksort::quicksort;

#[test]
fn test_quicksort() {
    let mut values = vec![12, 1, 5, 0, 6, 2];
    quicksort(&mut values);
    assert_eq!(values, vec![0, 1, 2, 5, 6, 12]);

    let mut values = vec![1, 13, 5, 10, 6, 2, 0];
    quicksort(&mut values);
    assert_eq!(values, vec![0, 1, 2, 5, 6, 10, 13]);
}
```

Looks a lot like unit tests, yes? The difference in an example like this is almost entirely semantics: in fact, this code sample includes unit tests which look nearly the same in listing 7.3:

Listing 7.3 Code sample of unit tests for quicksort implementation

```
#[cfg(test)]
mod tests {
    use crate::{partition, quicksort};

    #[test]
    fn test_partition() {
        let mut values = vec![0, 1, 2, 3];
        assert_eq!(
            partition(&mut values),
            (vec![0, 1, 2].as_mut_slice(), vec![3].as_mut_slice())
        );

        let mut values = vec![0, 1, 2, 4, 3];
        assert_eq!(
            partition(&mut values),
            (vec![0, 1, 2].as_mut_slice(), vec![3, 4].as_mut_slice())
        );
    }

    #[test]
    fn test_quicksort() {
        let mut values = vec![1, 5, 0, 6, 2];
        quicksort(&mut values);
        assert_eq!(values, vec![0, 1, 2, 5, 6]);

        let mut values = vec![1, 5, 10, 6, 2, 0];
        quicksort(&mut values);
        assert_eq!(values, vec![0, 1, 2, 5, 6, 10]);
    }
}
```

The only real difference is that in the unit tests we're also testing the `partition()` function, which is non-public.

Is this a case where we *shouldn't* bother writing integration tests? No. Why? Because we're creating a library with a public interface, and we should test the library as it's intended to be used *externally*. Integration tests live *outside* the library (or application) we're testing, thus they only have visibility to public (and external) interfaces. This forces us to write tests the same way that downstream users of our library or application would use the software.

Integration testing helps us make sure the public API works as intended from the perspective of external users.

7.2 Integration testing strategies

It wasn't too long ago that *test driven development*, or TDD, was all the rage. TDD is based on the idea that you write your tests *before* writing software. The theory of TDD is that writing tests first helps you build quality code faster. TDD seems to have fallen out of favour, but it does provide us with some insights, especially regarding integration testing.

One thing we can learn from TDD is that designing APIs is just as important as the testing itself. The ergonomics of your software matters, whether you're building libraries, command line

applications, web, desktop or mobile apps. The end user experience (or UX) is surfaced when writing integration tests; these tests force you to think about how your software is used from the perspective of the person using it.

Integration testing and unit testing aren't mutually exclusive; they should be used to complement each other where appropriate. Integration tests shouldn't be written the same way as unit tests, because we're testing different things. When we think about writing integration tests, we need to consider more than just the correctness of an algorithm or the logic it implements.

With integration tests we should think about them not only as a way of verifying our code works, but also as a way of testing the UX of our software. There are plenty of examples of good and bad software design, and the process of writing integration tests for your own software forces you to sample a taste of *your* design. It's easy to get tunnel vision and lose sight of the big picture when writing software, and integration tests are–by definition–a holistic perspective of your code.

I've personally experienced this tunnel vision problem many times. For example, when writing the dryoc crate I got a little carried away with some of the optional features, and it wasn't until I tried to write integration tests that I realized I'd done a poor job of designing the interface at the time. I had to refactor my design substantially to make the library easier to use.

Regarding TDD: *should* you write integration tests *before* writing your library or application? This is not a practice that I follow, but I don't believe it's necessarily bad, so you should figure out if it works for you then. I do think, however, that you should write integration tests in order to empathize with your end users. Which order you write the tests in is up to you. In any case, you should be flexible in your design and refactor mercilessly.

To quote a prolific architect and inventor:

> *When I am working on a problem, I never think about beauty but when I have finished, if the solution is not beautiful, I know it is wrong.*
>
> *– R. Buckminster Fuller*

The quicksort example above provides a good example of how we can improve our interface, which becomes apparent when writing tests for this library. Currently, we have a standalone `quicksort()` function which accepts a slice as input. This is fine, but we can make our code more Rustaceous by creating a trait and providing an implementation in listing 7.4:

Listing 7.4 Code listing for Quicksort trait

```
pub trait Quicksort {   ❶
    fn quicksort(&mut self) {}   ❷
}

impl<T: std::cmp::PartialOrd + Clone> Quicksort for [T] {   ❸
    fn quicksort(&mut self) {
        quicksort(self);   ❹
    }
}
```

❶ Here we define our public quicksort trait.

❷ We'll use the `quicksort()` method (rather than `sort()`) so we don't clash with the existing `sort()` method on `Vec` and slices.

❸ Here we define a generic implementation for our trait, which will work for any slice type that implements the `PartialOrd` and `Clone` traits.

❹ Here we just call our quicksort implementation directly.

Now, we can update our tests as shown in listing 7.5:

Listing 7.5 Code listing for integration test with Quicksort trait

```
#[test]
fn test_quicksort_trait() {
    use quicksort_trait::Quicksort;   ❶

    let mut values = vec![12, 1, 5, 0, 6, 2];
    values.quicksort();   ❷
    assert_eq!(values, vec![0, 1, 2, 5, 6, 12]);

    let mut values = vec![1, 13, 5, 10, 6, 2, 0];
    values.quicksort();   ❷
    assert_eq!(values, vec![0, 1, 2, 5, 6, 10, 13]);
}
```

❶ All we need to import is the `Quicksort` trait.

❷ Instead of `quicksort(&mut values)`, we can just write `values.quicksort()`.

This code doesn't look substantially different, and for the most part we've just using a bit of syntax sugar to clean things up. Calling `arr.quicksort()` instead of `quicksort(&mut arr)` looks nicer, and requires typing 4 fewer characters as we don't need to specify the explicit mutable borrow with `&mut`.

NOTE **We'll discuss traits in more detail in chapter 8.**

7.3 Built-in integration testing versus external integration testing

Rust's built-in integration testing will serve most people well, but it's not a panacea. You may benefit from external integration testing tools from time to time. For example, testing an HTTP service in Rust could be best served with simple (and nearly ubiquitous) tools like curl[32] or httpie[33]. These tools aren't related to Rust specifically, but rather they are generic tools that operate at the system level rather than the language level.

A quick Internet search will show there are many, many existing software test tools, especially for HTTP services, and unless you're trying to creating your own testing framework, it's almost always better to leverage existing tools rather than reinventing the wheel.

For command line applications written in Rust, writing the integration tests in Rust isn't always the best approach. Rust is designed for safety and performance: test harnesses don't usually need to be safe or fast, just correct. In many cases, it'll be much easier to write integration tests as Bash, Ruby, or Python script rather than a Rust program.

While it's great to do everything in Rust, you'll need to weigh the value in spending the time required to build your integration tests in Rust based on the complexity involved. Dynamic scripting languages have a lot of advantages for non-critical applications in that you can usually make things happen quickly with little effort even if you're an expert in Rust.

There is one big advantage to using Rust *only* for integration tests: you'll be able to run your tests on any platform supported by Rust, with no need for external tooling aside from the Rust toolchain. This can have some advantages especially in constrained environments. Additionally, if you find Rust to be your most productive language, then there's no reason not to use it.

7.4 Integration testing libraries and tooling

Most of the tools and libraries used for unit testing also apply for integration tests. There are, however, a few crates that can make life much easier for integration testing, which we'll explore in this section.

7.4.1 Using *`assert_cmd`* to test CLI applications

For testing command line applications, let's look at the `assert_cmd` [34] crate which makes it easy to run commands and check their result. To demonstrate, we'll create a command line interface for our quicksort implementation which sorts integers from CLI arguments, shown in listing 7.6:

Listing 7.6 Code listing of CLI application using quicksort

```
use std::env;

fn main() {
    use quicksort_proptest::Quicksort;

    let mut values: Vec<i64> = env::args()
        .skip(1)  ❶
        .map(|s| s.parse::<i64>().expect(&format!("{s}: bad input: ")))  ❷
        .collect();  ❸

    values.quicksort();

    println!("{values:?}");
}
```

❶ Reads the command line arguments, skipping the first argument, which is always the program name.

❷ Parses each value (a string) into an `i64`.

❸ Collects the values into a `Vec`.

We can test this by running `cargo run 5 4 3 2 1`, which will print `[1, 2, 3, 4, 5]`.

Now let's write some tests using `assert_cmd` in listing 7.7:

Listing 7.7 Code listing of quicksort CLI integration tests using `assert_cmd`

```
use assert_cmd::Command;

#[test]
fn test_no_args() -> Result<(), Box<dyn std::error::Error>> {  ❶
    let mut cmd = Command::cargo_bin("quicksort-cli")?;
    cmd.assert().success().stdout("[]\n");

    Ok(())  ❷
}

#[test]
fn test_cli_well_known() -> Result<(), Box<dyn std::error::Error>> {  ❶
    let mut cmd = Command::cargo_bin("quicksort-cli")?;
    cmd.args(&["14", "52", "1", "-195", "1582"])
        .assert()
        .success()
        .stdout("[-195, 1, 14, 52, 1582]\n");

    Ok(())  ❷
}
```

❶ Our test functions return a `Result`, which let's us use the `?` operator.

❷ At the end of the test we just return `Ok(())`. `()` is the special *unit* type, which can be used as a placeholder and has no value. It can be thought of as equivalent to a tuple with zero elements.

These tests are fine, but for testing against well-known values we can do a little better. Rather

than hardcoding into the source code, we can create some simple file-based fixtures to test against well-known values in a programmatic way.

First, we'll create a simple directory structure on the filesystem to store our test fixtures. The structure consists of numbered folders, with a file for the arguments (`args`) and the expected result (`expected`):

```
$ tree tests/fixtures
tests/fixtures
 1
    args
    expected
 2
    args
    expected
 3
     args
     expected

3 directories, 6 files
```

Next, we'll create a test that iterates over each directory within the tree and reads the arguments and expected result, then runs our test, and checks the result, as shown in listing 7.8:

Listing 7.8 Code listing of quicksort CLI integration tests with file-based fixtures

```
#[test]
fn test_cli_fixtures() -> Result<(), Box<dyn std::error::Error>> {
    use std::fs;
    let paths = fs::read_dir("tests/fixtures")?;  ❶

    for fixture in paths {  ❷
        let mut path = fixture?.path();
        path.push("args");  ❸
        let args: Vec<String> = fs::read_to_string(&path)?  ❹
            .trim()  ❹
            .split(' ')  ❹
            .map(str::to_owned)  ❹
            .collect();  ❹
        path.pop();  ❺
        path.push("expected");  ❻
        let expected = fs::read_to_string(&path)?;  ❼

        let mut cmd = Command::cargo_bin("quicksort-cli")?;  ❽
        cmd.args(args).assert().success().stdout(expected);  ❽
    }

    Ok(())
}
```

❶ Performs a directory listing within of `tests/fixtures` within our crate.

❷ Iterates over each listing within the directory.

❸ Pushes the `args` name into our path buffer.

❹ Reads the contents of the `args` file into a string and parses the string into a `Vec`. The `trim()` method removes the trailing newline from the `args` file, `split(' ')` will split the contents on spaces, `map(str::to_owned)` will convert a `&str` into an owned `String`, and finally `collect()` will collect the results into a `Vec`.

❺ Pop `args` off of the path buffer.

❻ Push `expected` onto the path buffer.

❼ Read the expected values from the file into a string.

❽ Lastly, run the quicksort CLI, pass the arguments, and check the expected results.

7.4.2 Using proptest with integration tests

Next, to make our tests even more robust, we can add the proptest crate (which we discussed in the previous chapter) to our quicksort implementation within an integration test in listing 7.9:

Listing 7.9 Code listing for proptest-based integration test with quicksort

```
use proptest::prelude::*;

proptest! {
    #[test]
    fn test_quicksort_proptest(
        vec in prop::collection::vec(prop::num::i64::ANY, 0..1000)
    ) {   ❶
        use quicksort_proptest::Quicksort;

        let mut vec_sorted = vec.clone();
        vec_sorted.sort();   ❷

        let mut vec_quicksorted = vec.clone();
        vec_quicksorted.quicksort();   ❸

        assert_eq!(vec_quicksorted, vec_sorted);
    }
}
```

❶ `prop::collection::vec` provides us with a `Vec` of random integers, with a length up to 1000.

❷ Here we clone then sort (using the built-in sorting method) the random values to use as our control.

❸ Here we clone and sort the random values using our quicksort implementation.

It's worth noting that testing with tools which automatically generate test data–such as proptest–can have unintented consequences should your tests have external side effects, such as making network requests or writing to an external database. You should try to design your tests to account for this, either by setting up and tearing down the whole environment before and after each test, or providing some other way to return to a known good state before and after the tests run. You may discover some surprisingly strange edge cases when using random data.

NOTE **The proptest crate prints the following warning when running as an integration test: `proptest: FileFailurePersistence::SourceParallel set, but failed to find lib.rs or main.rs`. This warning can ignored; refer to the GitHub issue at https://github.com/AltSysrq/proptest/issues/233 for more information.**

7.4.3 Other integration testing tools

Some more crates worth mentioning to turbocharge your integration tests are:

- rexpect[35]: automate and test interactive CLI applications
- `assert_fs` [36]: filesystem fixtures for applications that consume or produce files

7.5 Fuzz testing

Fuzz testing is similar to property testing, which we've already discussed in this chapter. The difference between fuzz testing and property testing, however, is that with fuzz testing you test your code with randomly generated data that *isn't necessarily valid*. When we do property testing, we generally restrict the set of inputs to values we consider valid. With property testing, we do this because it often doesn't makes sense to test all possible inputs, and we also don't have infinite time to test all possible combinations of input.

Fuzz testing, on the other hand, does away with the notion of valid and invalid, and just feeds random bytes into your code so you can see what happens. Fuzz testing is popular especially in security-sensitive contexts, where you want to understand what happens when code is misused.

A common example of this are public facing data sources, such as web forms. Web forms which can be filled with data from any source, which needs to be parsed, validated, and processed. Since these forms are out in the wild, there's nothing stopping someone from filling web forms with random data. For example, imagine a login form with a username and password, where someone (or something) could try every combination of username and password, or a list of the most common combinations, in order to gain access to the system either by guessing the correct combination, or by inject some magic set of bytes that causes an internal code failure and bypasses the authentication system. These type of vulnerabilities are *surprisingly* common, and fuzz testing is one strategy to mitigate them.

The main problem with fuzz testing is that it can take an infinite amount of time to test every set of possible inputs, but in practice you don't necessarily need to test *every* combination of input bits to find bugs with fuzz testing. In practice, you'll be quite surprised how quickly a fuzz test can find bugs in code you may have thought was bulletproof.

In order to fuzz test, we're going to use a library called libFuzzer[37], which is part of the LLVM project. You could use libFuzzer directly with FFI (we explored FFI in chapter 4), but instead we'll use a crate called cargo-fuzz which takes care of providing a Rust API for libFuzzer and generates boilerplate for us.

Before we dive into a code sample, let's talk about how libFuzzer works at a high level: the library will populate a structure (which you provide) with random data that contains function arguments, and it calls your code's function repeatedly. If the data triggers an error, this is detected by the library and a test case is constructed to trigger the bug.

Once we've installed the `cargo-fuzz` crate with `cargo install cargo-fuzz`, we can write a test. In listing 7.10 I've constructed a relatively simple function which looks like it works, but in fact it contains a subtle bug that will trigger under specific conditions:

Listing 7.10 Code listing for integer parsing function with a bug

```
pub fn parse_integer(s: &str) -> Option<i32> {
    // checks if string contains _only_ digits using a regular expression,
    // including negative numbers
    use regex::Regex;
    // will match a string with 1-10 digits, prefixed by an option `-`
    let re = Regex::new(r"^-?\d{1,10}$").expect("Parsing regex failed");
    if re.is_match(s) {
        Some(s.parse().expect("Parsing failed"))
    } else {
        None
    }
}
```

The function above will accept a string as input, and will parse the string into an `i32` integer provided it's between 1 and 10 digits in length, and optionally prefixed by a `-` (minus) symbol. If the input doesn't match the pattern, `None` is returned. The function shouldn't cause our program to crash on invalid input. Seems innocuous enough, however there's a major bug.

These kinds of bugs are surprisingly common, and something have all probably written at some point. And edge case bugs like this can lead to undefined behaviour, which–in a security context–can lead to bad things happening.

Next, let's create a small fuzz test using cargo-fuzz. First, we need to initialize the boilerplate code by running `cargo fuzz init`. This will create the following structure within our project:

```
$ tree .
.
 Cargo.lock
 Cargo.toml
 fuzz
    Cargo.lock
    Cargo.toml
    fuzz_targets
        fuzz_target_1.rs
 src
     lib.rs

3 directories, 6 files
```

Here we can see that cargo-fuzz created a new project in the `fuzz` subdirectory, and there's a test in `fuzz_target_1`. We can list the fuzz targets (or tests) with `cargo fuzz list`, which will print `fuzz_target_1`.

Next, we need to write the fuzz test. To test our function, we're just going to call it with a random string which is supplied by the fuzzing library. We'll use the arbitrary[38] crate to derive data in the form we need. The fuzz test is shown in listing 7.11:

Listing 7.11 Code listing of fuzz test

```
#![no_main]
use arbitrary::Arbitrary;
use libfuzzer_sys::fuzz_target;

#[derive(Arbitrary, Debug)]   ❶
struct Input {
    s: String,   ❷
}

fuzz_target!(|input: Input| {   ❸
    use fuzzme::parse_integer;

    parse_integer(&input.s);   ❹
});
```

❶ We use `derive` to automatically generate the `Arbitrary` and `Debug` traits for us.

❷ Our `Input` struct only contains one string, nothing else. The data in this struct will be populated arbitrarily by the fuzzer.

❸ Here we use the `fuzz_target!` macro provided by cargo-fuzz, which defines the entrypoint for our fuzz test.

❹ Here we just call our function with our random string data, which is provided by the fuzzer.

Now we're ready to run the fuzz test and see what happens. You probably already know at this point that there's a bug, so we expect it to crash. When we run the fuzzer with `cargo fuzz run fuzz_target_1` we'll see output that looks similar to listing 7.12 (which has been shortened because the fuzzer generates a lot of logging output):

Listing 7.12 Output of `cargo fuzz run fuzz_target_1`

```
cargo fuzz run fuzz_target_1
   Compiling fuzzme-fuzz v0.0.0
   (/Users/brenden/dev/code-like-a-pro-in-rust/code/c7/7.5/fuzzme/fuzz)
    Finished release [optimized] target(s) in 1.07s
    Finished release [optimized] target(s) in 0.01s
     Running `fuzz/target/x86_64-apple-darwin/release/fuzz_target_1
     -artifact_prefix=/Users/brenden/dev/code-like-a-pro-in-rust/
     code/c7/7.5/fuzzme/fuzz/artifacts/fuzz_target_1/
     ➥/Users/brenden/dev/code-like-a-pro-in-rust/code/c7/7.5/fuzzme/
     ➥fuzz/corpus/fuzz_target_1`
fuzz_target_1(14537,0x10d0a6600) malloc: nano zone abandoned due to
inability to preallocate reserved vm space.
INFO: Running with entropic power schedule (0xFF, 100).
... snip ...

Failing input:

    fuzz/artifacts/fuzz_target_1/
    crash-105eb7135ad863be4e095db6ffe64dc1b9a1a466

Output of `std::fmt::Debug`:

    Input {
        s: "8884844484",
    }

Reproduce with:

    cargo fuzz run fuzz_target_1 fuzz/artifacts/fuzz_target_1/
    crash-105eb7135ad863be4e095db6ffe64dc1b9a1a466

Minimize test case with:

    cargo fuzz tmin fuzz_target_1 fuzz/artifacts/fuzz_target_1/
    crash-105eb7135ad863be4e095db6ffe64dc1b9a1a466
```

NOTE **Running the fuzzer can take quite some time, even on fast machines. While this example should trigger fairly quickly (within 60 seconds in most cases), more complex tests may take much longer. For unbounded data (such as a string with no limit in length), the fuzzer can take an infinite amount of time.**

Near the bottom of the output, cargo-fuzz prints information about the input that caused the crash. Additionally, it creates a test case for us which we can use to make sure this bug isn't triggered again in the future. For the example above, we can simply run `cargo fuzz run fuzz_target_1 fuzz/artifacts/fuzz_target_1/crash-105eb7135ad863be4e095db6ffe64dc1b9a1a466` to test our code again with the same input, which will make it easy to test a fix for this bug without having to re-run the fuzzer from scratch. It can take a long time to find test cases that trigger a crash, so this is helpful in limiting the amount of time we need to spend running the fuzzer.

As an exercise for the reader, try modifying the function so it no longer crashes. There are a few different ways to solve this problem, and I'll provide a hint: the `parse()` method already returns

a `Result` for us.

For more details on using cargo-fuzz, consult the documentation at https://rust-fuzz.github.io/book/.

7.6 Summary

- Integration tests complement unit tests, but they differ in one major way: integration tests only apply to public interfaces.
- We can use integration tests as a way to test our API design and make sure it's appropriate and well designed for the end user.
- Rust's built in integration testing framework provides minimal features, but it's more than adequate for most purposes.
- Like unit tests, Rust's integration tests use the libtest library which is part of core Rust.
- Crates like proptest, `assert_cmd`, `assert_fs`, and rexpect can be used to further enhance our integration tests.
- The cargo-fuzz crate provides libFuzzer integration and Cargo commands to set up and run fuzz tests.

8 Async Rust

This chapter covers

- Thinking asynchronously: an overview of async programming
- Exploring Rust's async runtimes
- Handling async task results with futures
- Mixing sync and async
- Using the `async` & `.await` features
- Managing concurrency & paralellism with async
- Implementing an async observer
- When not to use async
- Tracing & debugging async code
- Dealing with async when testing

Concurrency is an important concept in computing, and it's one of the greatest force multipliers of computers. Concurrency allows us to process inputs and outputs–such as data, network connections, or peripherals–faster than we might be able to do without concurrency. And it's not always about speed, but also latency, overhead, and system complexity. We can run thousands or millions of tasks concurrently, as illustrated in figure 8.1, because concurrent tasks tend to be relatively light-weight. We can create, destroy, and manage many concurrent tasks with very little overhead.

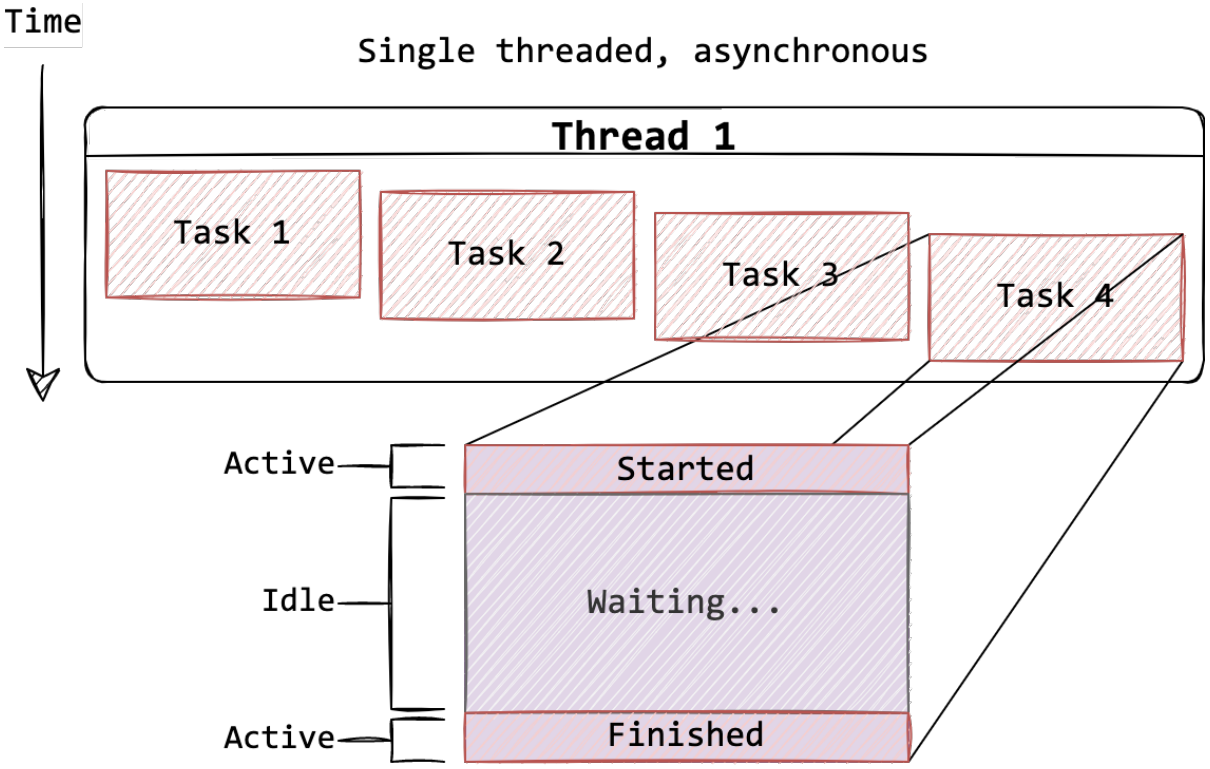

Figure 8.1 Tasks executing concurrently within the same thread

Asynchronous programming uses concurrency to take advantage of idle processing time between tasks. Some kinds of tasks, such as I/O, are much slower than ordinary CPU instructions, and after a slow task is started we can set it aside to work on other tasks while waiting for the slow tasks to complete.

Concurrency shouldn't be confused with *parallelism* (which I'll define here as the ability to execute multiple tasks simultaneously). Concurrency differs from parallelism in that tasks may be executed concurrently without being executed in parallel. In figure 8.2 I've illustrated

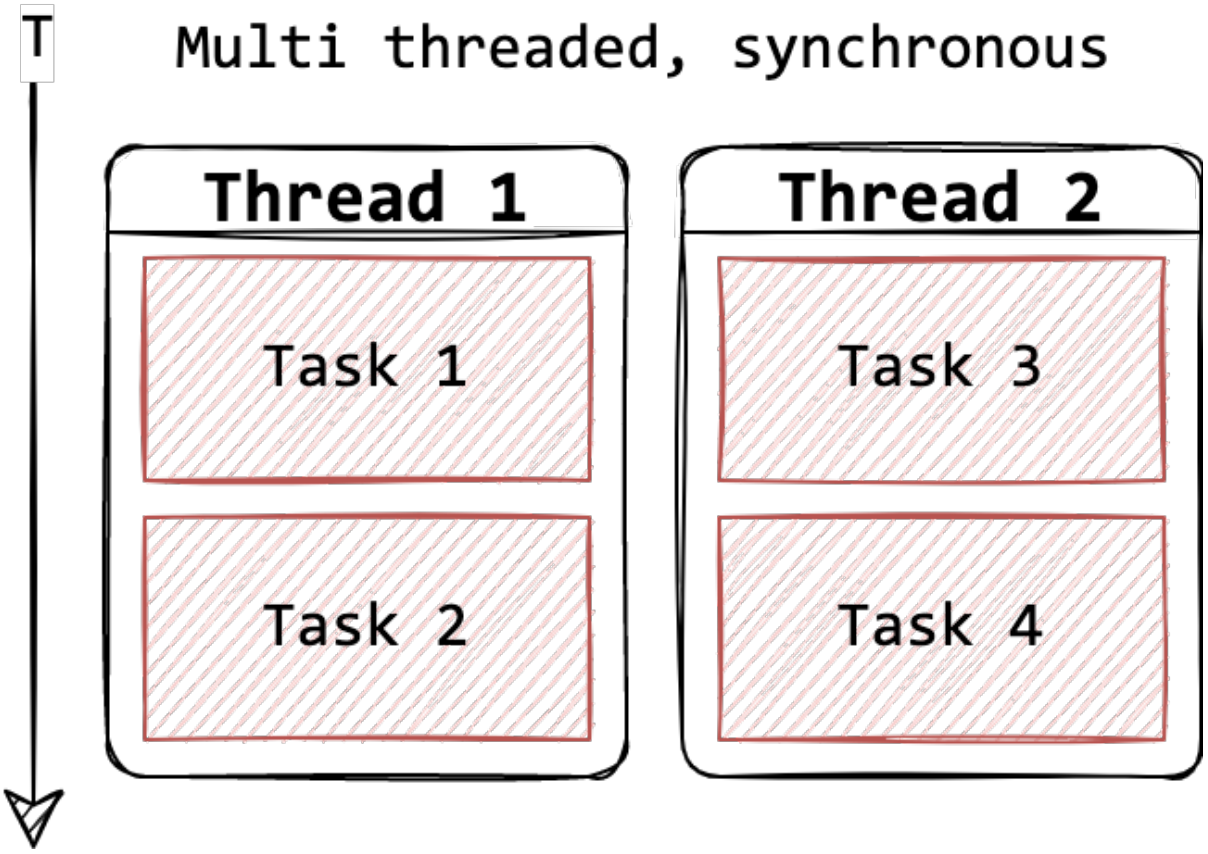

Figure 8.2 Synchronous tasks executing in parallel across 2 threads

To think about this analogously, consider how humans operate consciously: we can't do most tasks in parallel, but we can do a lot of things concurrently. For example, try having a conversation with 2 or more people at the same time: it's much harder than it sounds. It's possible to talk to many people at the same time, but you have to context switch between them, and pause when switching from one person to the other. Humans do concurrency reasonably well, but we suck at parallelism.

In this chapter, we're going to discuss Rust's asynchronous concurrency system which provides *both* concurrency and parallelism, depending on what your needs are. It's relatively easy to implement parallelism without async (using threads), but it's *quite hard* to implement concurrency without async. Async is a big topic, so we're really only going to cover the basics in this chapter. However, if you're already familiar with asynchronous programming, you'll find everything you need to be effective with async in Rust.

8.1 Runtimes

Rust's async is similar to what you may have encountered in other languages, though it has its own unique features in addition to borrowing much of what works well from other languages. If you're familiar with async from JavaScript, Python, or even C++'s `std::async`, you should have no problem adjusting to Rust's async. Rust does have one big difference: the language itself does not provide or prescribe an asynchronous runtime implementation. Rust *only* provides the `Future` trait, the `async` keyword, and `.await` statement: the implementation details are largely left to third-party libraries.

As of the time of writing, there are three widely used async runtime implementations, outlined in the table below:

Table 8.1 Summary of async runtimes

Name	Downloads[39]	Description
Tokio	59,642,490	Full-featured async runtime
async-std	8,075,376	Rust standard library implementation with async
smol	1,498,808	A lightweight runtime, intended to compete with Tokio

Both async-std and smol provide compatibility with the Tokio runtime, however it is *not* possible to mix competing async runtimes within the same context in Rust. As such, the recommendation is to use Tokio for most purposes as it is the most mature and widely used runtime. It may become easier to swap or interchange runtimes in the future, but for the time being this is not worth the headache.

Crates which provide async features could theoretically use any runtime, but in practice this is uncommon as they typically work best with one particular runtime. As such, some bifurcation exists in the Rust async ecosystem, where most crates are designed to work with Tokio, but some are specific to smol or async-std.

For the reasons outlined above, this book will focus on the Tokio async runtime as the preferred async runtime.

8.2 Thinking asynchronously

When we talk about *asynchronous programming*, we are usually referring to a way of handling control flow for any operation that involves waiting for a task to complete, which is often I/O. Examples of this are interacting with the file system or a socket, but it could also be slow operations such as computing a hash or waiting for a timer to finish. Most people are familiar with *synchronous* I/O, which (with the exception of certain languages like JavaScript) is the default mode of handling I/O. The main advantages of using asynchronous programming (as opposed to synchronous) are twofold:

- I/O tends to be very fast with async, because there is no need to perform a context switch between threads to support concurrency. Context switching, which often involves synchronization or locking with mutexes, can create a surprising amount of overhead.
- It's often much easier to reason about software that's written asynchronously because we don't have to worry as much about race conditions.
- Asynchronous tasks are very light-weight, thus we can easily handle thousands or millions of asynchronous tasks simultaneously.

When it comes to I/O operations in particular, the amount of time waiting for operations to complete is often *far* greater than the amount of time spent processing the result of the I/O operation. Because of this, we can do other work while we're waiting for tasks to finish, rather than executing every task sequentially. In other words, with async programming we are effectively breaking up and interleaving our function calls between the gaps created by time spent waiting for an I/O operation to finish.

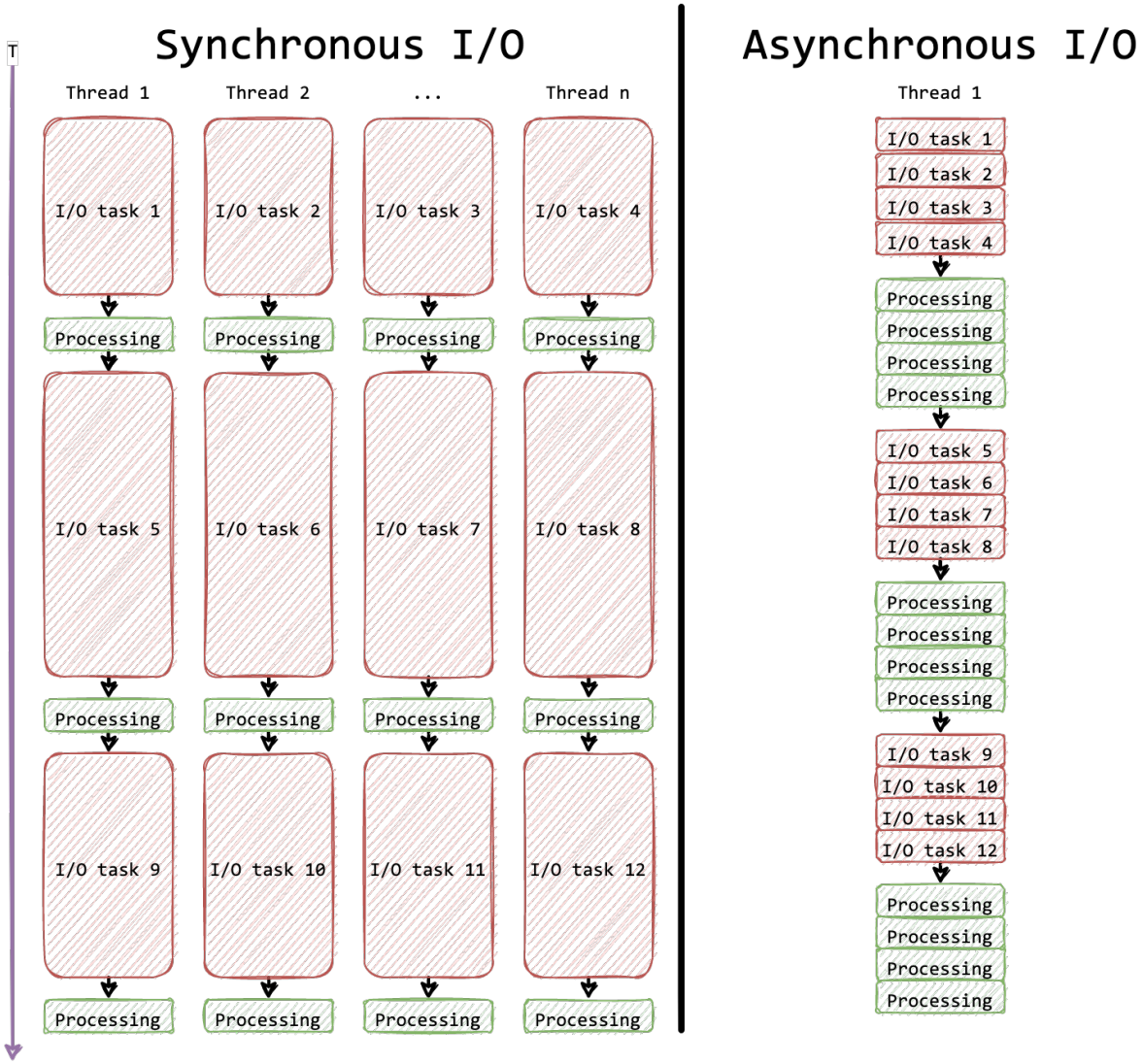

Figure 8.3 Comparing Synchronous to Asynchronous I/O

In figure 8.3, I've illustrated the difference with respect to T (time) of blocking versus non-blocking I/O operations. Async I/O is non-blocking, whereas synchronous I/O is blocking. If we assume the time to process the result of an I/O operation is much less than the time spent waiting for I/O to complete, asynchronous I/O will often be faster. It should also be noted that you *can* use multi-threaded programming with async, but often it's faster to simply use a single thread.

There's no free lunch here, however: if the time to process data from I/O operations becomes greater than the time spent waiting for I/O to complete, we'd see *worse* performance (assuming single-threaded async). As such, async isn't necessarily ideal for every use case. The good news is that Tokio provides quite a bit of flexibility in choosing how to execute async tasks, such as how many worker threads to use.

You can also mix parallelism with async, so comparing async directly to synchronous

programming is not always meaningful. Async code can run concurrently *in parallel* across several threads, which acts like a performance multiplier which synchronous code can't really compete with.

Once you adjust to the mental model required for thinking about async programming, it's much less complex than synchronous programming, especially when we're comparing it to multi-threaded synchronous programming.

8.3 Futures: handling async task results

Most async libraries and languages are based on *futures*, which is a design pattern for handling tasks that return a result in the future (hence the name). When we perform an asynchronous operation, the result of that operation is a future, as opposed to directly returning the value of the operation itself (as we'd see in synchronous programming, or an ordinary function call). While futures are a convenient abstraction, they do require a little more work on behalf of the programmer to handle them correctly.

To better understand futures, let's consider how a timer works: we can create (or start) an async timer which returns a future to signal the completion of the timer. Merely *creating* the timer is not enough: we also need to tell the executor (which is part of the async runtime) to execute the task. In synchronous code, when we want to sleep for 1 second, we can just call the `sleep()` function.

> **NOTE** **It's true that you could call `sleep()` within async code, but you should never do this. An important rule of asynchronous programming is to never block the main thread. While calling `sleep()` won't cause your program to crash, it will effectively defeat the purpose of async programming and is considered and anti-pattern.**

To compare an async timer to synchronous one, let's look at what it takes to write a tiny Rust program that sleeps for 1 second and prints "Hello, world!". First, let's look at the synchronous code:

```
fn main() {
    use std::{thread, time};

    let duration = time::Duration::from_secs(1);

    thread::sleep(duration);

    println!("Hello, world!");
}
```

The synchronous code looks nice and simple. Next, let's examine the async version:

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

    let duration = time::Duration::from_secs(1);

    tokio::runtime::Builder::new_current_thread()
        .enable_time()  ❶
        .build()
        .unwrap()
        .block_on(async {  ❷
            tokio::time::sleep(duration).await;
            println!("Hello, world!");
        });
}
```

❶ The runtime supports time or I/O, which can be enabled individually, or entirely with `enable_all()`.

❷ We create an async block, which waits on the future returned by `tokio::time::sleep()` and then prints "Hello, world!".

Yikes! That's much more complicated. Why do we need all this complexity? In short, async programming requires special control flow, which is mostly managed by the runtime, but still requires a different style of programming. The runtime's scheduler decides what to run when, but we need to yield to the scheduler to allow an opportunity for it to switch between tasks. The runtime will manage most of the details, but we still need to be aware of this to use async effectively. Yielding to the scheduler (in most cases) is as simple as using `.await`, which we'll discuss more in the next section.

SIDEBAR **What does it mean to block the main thread?**

As I've already alluded, the trick to writing good async code is to avoid blocking the main thread. When we say "block the main thread", what we really mean is that the runtime should not be prevented from switching tasks for long periods of time. We typically consider I/O to be a blocking operation, because the amount of time an I/O operation takes to complete depends on various factors outside the context of our program and its control. But you could also have strictly CPU-bound tasks that are considered blocking, provided they take long enough to complete.

We can prevent blocking the main thread for too long by introducing yield points. A yield point is any code that passes control back to the scheduler. Joining or waiting on a future creates a yield point by passing control up through the chain to the runtime.

The question of what constitutes a long period of time is largely context dependent, so I can't provide hard guidelines. We can, however, estimate what constitute fast versus slow operations by looking at how long CPU-bound and I/O-bound operations typically take. To gauge the difference between fast and slow, we can compare a typical function call (which is a fast operation) to how long a simple I/O operation takes to (a slow operation).

Let's estimate the time it takes for a typical function to execute. We can calculate the time of 1 clock cycle by taking the inverse of the CPU frequency (assuming 1 instruction per clock cycle).

For example, for a 2GHz CPU, the time per instruction is 0.5 ns. For an operation that requires 50 instructions (which would approximate a typical function call), we can assume about 25 ns to execute.

By comparison, a small I/O operation such as reading 1024 bytes from a file can take significantly longer. Running a small test on my laptop we can demonstrate this:

```
$ dd if=/dev/random of=testfile bs=1k count=1   ❶
1+0 records in
1+0 records out
1024 bytes (1.0 kB, 1.0 KiB) copied, 0.000296943 s, 3.4 MB/s
$ dd if=testfile of=/dev/null   ❷
2+0 records in
2+0 records out
1024 bytes (1.0 kB, 1.0 KiB) copied, 0.000261919 s, 3.9 MB/s
```

❶ Writes 1k of random bytes from `/dev/null` to `testfile`.

❷ Reads the contents of `testfile` and writes them to `/dev/null`.

In the test above, reading from a small file takes on the order of 262 s, which is about 5,240 times longer than 50 ns. Network operations are likely to be another 1-2 orders of magnitude slower, depending on a number of factors.

For non-I/O operations which you think might take a relatively long time to complete, you should either treat them as blocking using `tokio::task::spawn_blocking()`, or break them up by introducing `.await` as needed, allowing the scheduler has an opportunity to give other tasks time to run. If unsure, you should benchmark your code to decide whether or not you would benefit from such optimizations.

8.3.1 Defining a runtime with *`#[tokio::main]`*

Tokio provides a macro for wrapping our `main()` function, so we can simplify the timer code above into the following form if we want:

```
#[tokio::main]
async fn main() {
    use std::time;

    let duration = time::Duration::from_secs(1);

    tokio::time::sleep(duration).await;
    println!("Hello, world!");
}
```

With the help of some syntax sugar, our code now looks just like the synchronous version. We've turned `main()` into an async function with the `async` keyword, and the `#[tokio::main]`

handles the boilerplate needed to start the Tokio runtime and create the async context we need.

Remember that the result of any async task is a future, but we need to execute that future on the runtime *before* anything actually happens. In Rust this is normally done with the `.await` statement, which we will discuss in the next section.

8.4 async & .await: when and where

The `async` and `.await` keywords are quite new in Rust. It's possible to use futures directly *without* these, but you're better off just using `async` and `.await` when possible because they handle much of the boilerplate without sacrificing functionality. A function or block of code marked as `async` will return a future, and `.await` tells the runtime we want to wait for a result. The `async` & `.await` syntax allows us to write async code that looks like synchronous code, but without much of the complexity that comes with working with futures. You can use `async` with functions, closures, and code blocks. `async` blocks don't execute until they're polled, which you can do with `.await`.

Under the hood, using `.await` on a future uses the runtime to call the `poll()` method from the `Future` trait, and waits for the future's result. If you never call `.await` (or explicitly poll a future), the future will never execute.

To use `.await`, we need to be within an async context. We can create an async context by creating a block of code marked with `async`, and executing that code on the async runtime. You don't *have* to use `async` and `.await`, but it's much easier to do things this way, and the Tokio runtime (along with many other async crates) have been designed to be used this way.

For example, consider the following program which includes a fire-and-forget async code block spawned with `tokio::task::spawn()`:

```
#[tokio::main]
async fn main() {
    async {
        println!("This line prints first");
    }
    .await;
    let _future = async {
        println!("This line never prints");
    };
    tokio::task::spawn(async {
        println!(
            "This line prints sometimes, depending on how quick it runs"
        )
    });

    println!("This line always prints, but it may or may not be last");
}
```

If you run the code above repeatedly, it will (confusingly) print *either* 2 or 3 lines. The first `println!()` will *always* print before the others, because of the `.await` statement which awaits the result of the first future. The second `println!()` never prints because we didn't execute the

future by calling `.await` or spawning it on the runtime. The third `println!()` is spawned onto the Tokio runtime, but we don't wait for it to complete, so it's not guaranteed to run and we don't know if it will run before or after the last `println!()`, which will always print.

Why does the 3rd `println!()` not print consistently? It's possible that the program will exit before the Tokio runtime's scheduler gets a chance to execute the code. If we want to guarantee that the code runs before exiting, we need to wait for the future returned by `tokio::task::spawn()` to complete.

The `tokio::task::spawn()` function has another important feature: it allows us to launch an async task on the async runtime from *outside* an async context. It also returns a future (`tokio::task::JoinHandle`, specifically) which we can pass around like any other object. Tokio's join handles also allow us to abort tasks if we want. Let's look at the example in listing 8.1:

Listing 8.1 Spawning a task with `tokio::task::spawn()`

```
use tokio::task::JoinHandle;

fn not_an_async_function() -> JoinHandle<()> {   ❶
    tokio::task::spawn(async {   ❷
        println!("Second print statement");
    })
}

#[tokio::main]
async fn main() {
    println!("First print statement");
    not_an_async_function().await.ok();   ❸
}
```

❶ A normal function, returning a `JoinHandle` (which implements the `Future` trait).

❷ Our `println!()` task is spawned on the runtime.

❸ We use `.await` on the future returned from our function to wait for it to execute.

In the code above, we've created a normal function that returns a `JoinHandle` (which is just a type of future). `tokio::task::spawn()` returns a `JoinHandle` which allows us to join the task (i.e., retrieve the result of our code block, which is just a unit in this example).

What happens if we want to use `.await` *outside* of an async context? Well, in short, you can't. You *can*, however, block on the result of a future to await its result using the `tokio::runtime::Handle::block_on()` method. To do so, you'll need to obtain a *handle* for the runtime, and move that runtime handle into the thread where you want to block. Handles can be cloned and shared, providing access to the async runtime from outside an async context, as shown in listing 8.2:

Listing 8.2 Using a Tokio `Handle` to spawn tasks

```
use tokio::runtime::Handle;

fn not_an_async_function(handle: Handle) {
    handle.block_on(async {   ❶
        println!("Second print statement");
    })
}

#[tokio::main]
async fn main() {
    println!("First print statement");

    let handle = Handle::current();   ❷
    std::thread::spawn(move || {   ❸
        not_an_async_function(handle);   ❹
    });
}
```

❶ Spawns a blocking async task on our runtime using a runtime handle.

❷ Gets the runtime handle for the current runtime context.

❸ Spawns a new thread, capturing variables with a move.

❹ Calls our non-async function in a separate thread, passing the async runtime handle along.

That's not pretty, but it works. There are a few cases where you *might* want to do things like this, which we'll discuss in the next section, but for the most part you should try and use `async` and `.await` when possible.

In short: wrap code blocks (including functions and closures) with `async` when you want to perform async tasks or return a future, and use `.await` when you need to wait for an async task. Creating an async block *does not* execute the future: it still needs to be executed (or spawned with `tokio::task::spawn()`) on the runtime.

8.5 Concurrency & parallelism with async

At the beginning of the chapter I discussed the differences between concurrency and parallelism. With async we don't get either concurrency or parallelism for free. We still have to think about how to structure our code to take advantage of these features.

With Tokio, there's no explicit control over parallelism (aside from launching a blocking task with `tokio::task::spawn_blocking()`, which always runs in a separate thread). We *do* have explicit control over concurrency, but we can't control the parallelism of individual tasks, as those details are left up to the runtime. Tokio *does* allow us to configure is the number of worker threads, but the runtime will decide which threads to use for each task.

Introducing concurrency into our code can be accomplished one of three ways:

- Spawning tasks with `tokio::task::spawn()`
- Joining multiple futures with `tokio::join!(…)` or `futures::future::join_all()`
- Using the `tokio::select! { … }` macro, which allows us to wait on multiple concurrent code branches (modeled after the UNIX `select()` system call)

To introduce *parallelism*, we have to use `tokio::task::spawn()`, but we don't get *explicit* parallelism this way. Instead, when we spawn a task we're telling Tokio that this task can be executed on any thread, but Tokio still decides which thread to use. If we launch our Tokio runtime with only 1 worker thread, for example, all tasks will execute in one thread even when we use `tokio::task::spawn()`.

We can demonstrate the behaviour with some sample code, shown in listing 8.3:

Listing 8.3 Demonstrating async concurrency and parallelism

```
async fn sleep_1s_blocking(task: &str) {
    use std::{thread, time::Duration};
    println!("Entering sleep_1s_blocking({task})");
    thread::sleep(Duration::from_secs(1));   ❶
    println!("Returning from sleep_1s_blocking({task})");
}

#[tokio::main(flavor = "multi_thread", worker_threads = 2)]   ❷
async fn main() {
    println!("Test 1: Run 2 async tasks sequentially");
    sleep_1s_blocking("Task 1").await;   ❸
    sleep_1s_blocking("Task 2").await;

    println!("Test 2: Run 2 async tasks concurrently (same thread)");
    tokio::join!(   ❹
        sleep_1s_blocking("Task 3"),
        sleep_1s_blocking("Task 4")
    );

    println!("Test 3: Run 2 async tasks in parallel");
    tokio::join!(   ❺
        tokio::spawn(sleep_1s_blocking("Task 5")),
        tokio::spawn(sleep_1s_blocking("Task 6"))
    );
}
```

❶ Here we intentionally use `std::thread::sleep()`, which is blocking, in order to demonstrate parallelism.

❷ We're explicitly configuring Tokio with 2 worker threads, which allows us to run tasks in parallel.

❸ Here we call our `sleep_1s()` function twice sequentially, with no concurrency or parallelism.

❹ Here we call `sleep_1s()` twice using `tokio::join!()`, which introduces concurrency.

❺ Lastly, we're *spawning* our `sleep_1s()`, and then joining on the result, which introduces parallelism.

Running the code above will generate the following output:

```
Test 1: Run 2 async tasks sequentially
Entering sleep_1s_blocking(Task 1)
Returning from sleep_1s_blocking(Task 1)
Entering sleep_1s_blocking(Task 2)
Returning from sleep_1s_blocking(Task 2)
Test 2: Run 2 async tasks concurrently (same thread)
Entering sleep_1s_blocking(Task 3)
Returning from sleep_1s_blocking(Task 3)
Entering sleep_1s_blocking(Task 4)
Returning from sleep_1s_blocking(Task 4)
Test 3: Run 2 async tasks in parallel
Entering sleep_1s_blocking(Task 5)   (1)
Entering sleep_1s_blocking(Task 6)
Returning from sleep_1s_blocking(Task 5)
Returning from sleep_1s_blocking(Task 6)
```

(1) In the 3rd test, we can see our `sleep_1s()` function is running in parallel because both functions are entered before returning.

From the output above, we can see that only the 3rd test, where we launch each task with `tokio::spawn()` (which is equivalent to `tokio::task::spawn()`), does the code executes in parallel. We can tell it's executing in parallel because we see both of the `Entering ...` statements *before* the `Returning ...` statements. An illustration of the sequence of events is shown in figure 8.4.

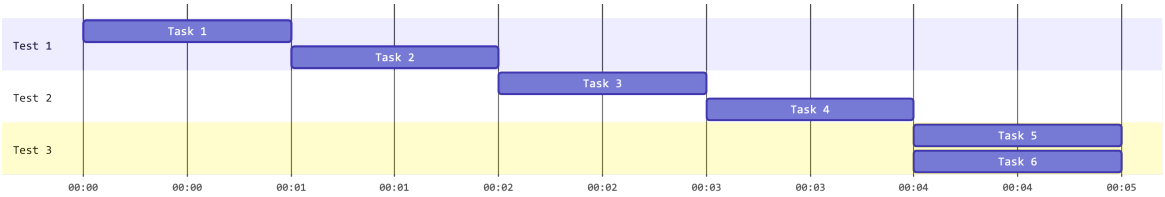

Figure 8.4 Diagram showing sequence of events with blocking sleep

Note that, while the 2nd test is indeed running concurrently, it is not running in parallel, thus the tasks execute sequentially because we used a *blocking* sleep in listing 8.3. Let's update the code to add non-blocking sleep as follows:

```
async fn sleep_1s_nonblocking(task: &str) {
    use tokio::time::{sleep, Duration};
    println!("Entering sleep_1s_nonblocking({task})");
    sleep(Duration::from_secs(1)).await;
    println!("Returning from sleep_1s_nonblocking({task})");
}
```

After updating our `main()` to add 3 tests with the non-blocking sleep, we get the following output:

```
Test 4: Run 2 async tasks sequentially (non-blocking)
Entering sleep_1s_nonblocking(Task 7)
Returning from sleep_1s_nonblocking(Task 7)
Entering sleep_1s_nonblocking(Task 8)
Returning from sleep_1s_nonblocking(Task 8)
Test 5: Run 2 async tasks concurrently (same thread, non-blocking)
Entering sleep_1s_nonblocking(Task 9)  (1)
Entering sleep_1s_nonblocking(Task 10)
Returning from sleep_1s_nonblocking(Task 10)
Returning from sleep_1s_nonblocking(Task 9)
Test 6: Run 2 async tasks in parallel (non-blocking)
Entering sleep_1s_nonblocking(Task 11)
Entering sleep_1s_nonblocking(Task 12)
Returning from sleep_1s_nonblocking(Task 12)
Returning from sleep_1s_nonblocking(Task 11)
```

(1) We can see here that our sleep happens concurrently now that we changed the sleep function to non-blocking.

Figure 8.5 illustrates how both test 5 and 6 appear to execute in parallel, although only test 6 is actually running in parallel, whereas test 5 is running concurrently.

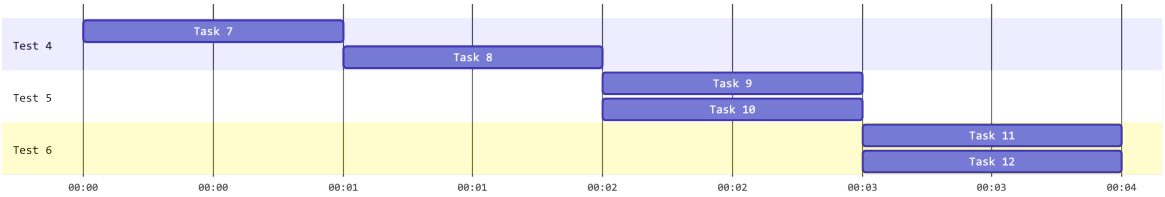

Figure 8.5 Diagram showing sequence of events with non-blocking sleep

If we again update our Tokio settings and change `worker_threads = 1`, then re-run the test, you will see in the blocking sleep version *all* of the tasks run sequentially, but in the concurrent non-blocking version they still run concurrently even with 1 thread.

It may take some time to wrap your head around concurrency and parallelism with async Rust, so don't worry if this seems confusing at first. I recommend trying this sample yourself and experimenting with different parameters to get a better understanding of what's going on.

8.6 Implementing an async observer

In chapter 9 we implemented the observer pattern. This pattern also happens to be *incredibly* useful in async programming, so we'll revisit that example again, but this time we'll make it work with async.

At the time of writing, there is one big limitation of Rust's async support: we can't use traits with async methods. For example, the following code is invalid:

```
trait MyAsyncTrait {
    async fn do_thing();
}
```

Because of this, it makes implementing the observer pattern with async somewhat tricky. There are a few ways to work around this problem, but I will present one solution which also provides some insight into how Rust implements the `async fn` syntax sugar.

As mentioned earlier in this chapter, the `async` and `.await` features are just convenient syntax for working with futures. When we declare an `async` function or code block, the compiler is wrapping that code with a future for us. Thus, we can still create the equivalent of an `async` function with traits, but we have to do it explicitly (without the syntax sugar).

The observer trait we defined back in chapter 9 looked as follows:

```
pub trait Observer {
    type Subject;
    fn observe(&self, subject: &Self::Subject);
}
```

To convert the `observe()` method into an async function, the first step is to make it return a `Future`. We can try something like this as a first step:

```
pub trait Observer {
    type Subject;
    type Output: Future<Output = ()>;   ❶
    fn observe(&self, subject: &Self::Subject) -> Self::Output;   ❷
}
```

❶ Here we define an associated type with the `Future` trait bound, returning `()`.

❷ Now our `observe()` method returns the `Output` associated type.

At first glance this seems like it should work, and the code compiles. However, as soon as we try to implement the trait we'll run into a few issues. For one, because `Future` is just a trait (not a concrete type), we don't know what type to specify for `Output`. Thus, we *can't* use an associated type this way. Instead, we need to use a trait object. To do this, we have to return our future within a `Box`. We'll update the trait like so:

```
pub trait Observer {
    type Subject;
    type Output;   ❶
    fn observe(
        &self,
        subject: &Self::Subject,
    ) -> Box<dyn Future<Output = Self::Output>>;   ❷
}
```

❶ We kept the associated type for the return type here, which adds some flexibility.

❷ Now we return a `Box<dyn Future>` instead.

Let's try to put it together by implementing our new async observer for `MyObserver`:

```
struct Subject;
struct MyObserver;

impl Observer for MyObserver {
    type Subject = Subject;
    type Output = ();
    fn observe(
        &self,
        _subject: &Self::Subject,
    ) -> Box<dyn Future<Output = Self::Output>> {
        Box::new(async {  ❶
            // do some async stuff here!
            use tokio::time::{sleep, Duration};
            sleep(Duration::from_millis(100)).await;
        })
    }
}
```

❶ Note that we have to box the future we're returning.

So far so good, the compiler is happy too. Now what happens if we try to test it? Let's write a quick test:

```
#[tokio::main]
async fn main() {
    let subject = Subject;
    let observer = MyObserver;
    observer.observe(&subject).await;
}
```

And now we hit our next snag. Trying to compile this will generate the following error:

```
error[E0277]: `dyn Future<Output = ()>` cannot be unpinned
  --> src/main.rs:29:31
   |
29 |     observer.observe(&subject).await;
   |                                ^^^^^^ the trait `Unpin` is not
   ➥ implemented for `dyn Future<Output = ()>`
   |
   = note: consider using `Box::pin`
   = note: required because of the requirements on the impl of `Future` for
   ➥ `Box<dyn Future<Output = ()>>`
   = note: required because of the requirements on the impl of `IntoFuture`
   ➥ for `Box<dyn Future<Output = ()>>`
help: remove the `.await`
   |
29 -     observer.observe(&subject).await;
29 +     observer.observe(&subject);
   |

For more information about this error, try `rustc --explain E0277`.
```

What's happening here? To understand, we need to look at the `Future` trait from the Rust standard library:

```
pub trait Future {
    type Output;
    fn poll(self: Pin<&mut Self>, cx: &mut Context<'_>) -> Poll<Self::Output>;
}
```

Notice that, the `poll()` method takes its `self` parameter as the type `Pin<&mut Self>`. In other words, before we can poll a future (which is what `.await` does), it needs to be *pinned*. A pinned pointer is a special kind of pointer in Rust that can't be moved (until it's unpinned). Lucky for us, obtaining a pinned pointer is easy, we just need to update our `Observer` trait again as follows:

```
pub trait Observer {
    type Subject;
    type Output;
    fn observe(
        &self,
        subject: &Self::Subject,
    ) -> Pin<Box<dyn Future<Output = Self::Output>>>;   ❶
}
```

❶ Now we wrap our `Box` in `Pin`, which gives us a pinned box.

Next, we'll update our implementation like so:

```
impl Observer for MyObserver {
    type Subject = Subject;
    type Output = ();
    fn observe(
        &self,
        _subject: &Self::Subject,
    ) -> Pin<Box<dyn Future<Output = Self::Output>>> {   ❶
        Box::pin(async {   ❷
            // do some async stuff here!
            use tokio::time::{sleep, Duration};
            sleep(Duration::from_millis(100)).await;
        })
    }
}
```

❶ Now we return `Pin<Box<…>>`.

❷ `Box::pin()` conveniently returns a pinned box for us.

At this point our code will compile, and it works. You might think we're out of the woods, but unfortunately we are not. The implementation for the `Observable` trait is even *more* complicated. Let's take a look at that trait from chapter 9:

```
pub trait Observable {
    type Observer;
    fn update(&self);
    fn attach(&mut self, observer: Self::Observer);
    fn detach(&mut self, observer: Self::Observer);
}
```

We need to make the `update()` method from `Observable` async, but it's more complicated

because inside `update()` we pass `self` to each of the observers. Passing a self-reference inside an async method won't work without specifying a lifetime for that reference. Additionally, we need each `Observer` instance to implement both `Send` and `Sync` because we want to observe updates concurrently, which requires that our observers can move across threads.

The final form of our `Observer` trait is shown in listing 8.4:

Listing 8.4 Implementing the async `Observer` trait

```
pub trait Observer: Send + Sync {   ❶
    type Subject;
    type Output;
    fn observe<'a>(   ❷
        &'a self,   ❸
        subject: &'a Self::Subject,   ❹
    ) -> Pin<Box<dyn Future<Output = Self::Output> + 'a + Send>>;   ❺
}
```

❶ We add the `Send + Sync` supertraits to make sure our observers can be used concurrently across threads.

❷ The `'a` lifetime allows us to pass `self` and `subject` as references.

❸ Here we apply `'a` to the `self` reference.

❹ Here we apply `'a` to the `subject` reference.

❺ We add `'a + Send` to the trait bounds to allow moving across threads, and make sure the return future is doesn't outlive any captured references for `'a`.

And our updated `Observable` trait is shown in listing 8.5:

Listing 8.5 Implementing the async `Observable` trait

```
pub trait Observable {
    type Observer;
    fn update<'a>(   ❶
        &'a self,
    ) -> Pin<Box<dyn Future<Output = ()> + 'a + Send>>;
    fn attach(&mut self, observer: Self::Observer);
    fn detach(&mut self, observer: Self::Observer);
}
```

❶ Like with `Observer`, we need to add a lifetime for our references.

And now we'll implement `Observable` for our `Subject` in listing 8.6:

Listing 8.6 Implementing the async `Observable` trait for `Subject`

```
pub struct Subject {
    observers:
        Vec<Weak<dyn Observer<Subject = Self, Output = ()>>>,
    state: String,
}

impl Subject {
    pub fn new(state: &str) -> Self {
        Self {
            observers: vec![],
            state: state.into(),
        }
    }

    pub fn state(&self) -> &str {
        self.state.as_ref()
    }
}

impl Observable for Subject {
    type Observer =
        Arc<dyn Observer<Subject = Self, Output = ()>>;
    fn update<'a>(&'a self) -> Pin<Box<dyn Future<Output = ()> + 'a + Send>>
    {
        let observers: Vec<_> =
            self.observers.iter().flat_map(|o| o.upgrade()).collect();  ❶

        Box::pin(async move {  ❷
            futures::future::join_all(  ❸
                observers.iter().map(|o| o.observe(self)),  ❹
            )
            .await;  ❺
        })
    }
    fn attach(&mut self, observer: Self::Observer) {
        self.observers.push(Arc::downgrade(&observer));
    }
    fn detach(&mut self, observer: Self::Observer) {
        self.observers
            .retain(|f| !f.ptr_eq(&Arc::downgrade(&observer)));
    }
}
```

❶ We generate the list of observers to notify *outside* the async context, and collect this into a new `Vec`.

❷ We use a `move` on our async block to move the captured `observers` list into the async block.

❸ Using `join_all()` here introduces concurrency across our observers.

❹ Each observer's `observe` function is called with the same `self` reference.

❺ Lastly, we `.await` on the join operation within our async block.

Now we can finally test our async observer pattern as shown in listing 8.7:

Listing 8.7 Testing our async observer pattern

```
#[tokio::main]
async fn main() {
    let mut subject = Subject::new("some subject state");

    let observer1 = MyObserver::new("observer1");
    let observer2 = MyObserver::new("observer2");

    subject.attach(observer1.clone());
    subject.attach(observer2.clone());

    // ... do something here ...

    subject.update().await;
}
```

Running the code above will produce the following output, exactly like it did from chapter 9:

```
observed subject with state="some subject state" in observer1
observed subject with state="some subject state" in observer2
```

8.7 Mixing sync and async

The Rust async ecosystem is growing fast, and many libraries have support for `async` and `.await`. However, in spite of this, there are cases where you may need to deal with synchronous and asynchronous code together. We already demonstrated 2 such examples in the previous section, but let's elaborate more on that.

YOUR TURN **As a general rule, you should avoid mixing sync and async. In some cases it may be worth the effort to add async support when it's missing, or upgrade code that's using older versions of Tokio which don't work with the new `async` and `.await` syntax.**

The most common reason you'll have to mix sync and async is when you're using a crate which doesn't support async, such as a database driver or networking library. For example, if you want to write an HTTP service using the Rocket crate[40] with async, you may need to read or write to a database that doesn't yet have async support. Adding async support to sufficiently complex libraries might not be the best use of your time, even when it's a noble cause.

To call synchronous code from within an async context, the preferred way is to use the `tokio::task::spawn_blocking()` function, which accepts a function and returns a future. When a call to `spawn_blocking()` is placed, it will execute the function provided on a thread queue managed by Tokio (which can be configured). You can then use `.await` on the future returned by `spawn_blocking()`, like you normally would with any async code.

Let's look at an example of `spawn_blocking()` in action, by creating code that writes a file asynchronously, and then reads it back sychronously:

```
use tokio::io::{self, AsyncWriteExt};

async fn write_file(filename: &str) -> io::Result<()> {   ❶
    let mut f = tokio::fs::File::create(filename).await?;
    f.write(b"Hello, file!").await?;
    f.flush().await?;

    Ok(())
}

fn read_file(filename: &str) -> io::Result<String> {   ❷
    std::fs::read_to_string(filename)
}

#[tokio::main]
async fn main() -> io::Result<()> {
    let filename = "mixed-sync-async.txt";
    write_file(filename).await?;

    let contents =
        tokio::task::spawn_blocking(|| read_file(filename)).await??;   ❸

    println!("File contents: {}", contents);

    tokio::fs::remove_file(filename).await?;

    Ok(())
}
```

❶ Writes "Hello, file!" to our file asynchronously.

❷ Reads the contents of our file into a string, returning the string, synchronously.

❸ Note the double `??`, because both `spawn_blocking()` and `read_file()` return a `Result`.

In the code above, we perform our synchronous I/O within the function called by `spawn_blocking()`. We can await the result just like any other ordinary async block, *except* that it's actually being executed on a separate blocking thread managed by Tokio. We don't have to worry about the implementation details, except that there needs to be an adequate number of threads allocated by Tokio. In the example above we just use the default values, but you can change the number of blocking threads with the Tokio runtime builder (for which the full list of parameters can be found at https://docs.rs/tokio/latest/tokio/runtime/struct.Builder.html).

SIDEBAR **Synchronizing async code**

Sometimes we need to synchronize async code, such as when we need to pass messages between different objects. Because our async code blocks may run across separate threads of execution, sharing data between them can be tricky, as we can introduce race conditions if we try to access data improperly (additionally, the Rust language won't allow it). An easy way to share data is to use shared state behind a mutex, but Tokio provides some better ways to share state.

Within its `sync` module, Tokio provides a number of tools for synchronizing async code. Notably, you will likely want to learn about the multi-producer single-consumer channel, which can be found within the `tokio::sync::mpsc` module. An `mpsc` channel lets you safely pass messages from multiple producers to a single consumer within an async context, without the need for explicit locking (i.e., introducing mutexes). Tokio provides other channels types, including `broadcast`, `oneshot`, and `watch`.

With `mpsc` channels you can build scalable concurrent message passing interfaces in async Rust without explicit locking. An `mpsc` channel can be unbounded, or bounded with a fixed length providing backpressure to producers.

Tokio's channels are not unlike what you may find in other actor or event processing frameworks, except they're relatively low-level, and fairly general-purpose. They're more similar to socket programming than what you might find in higher-level actor libraries.

For details on Tokio's synchronous tooling, refer to the `sync` module at https://docs.rs/tokio/latest/tokio/sync/index.html.

For the opposite case, of using async code within synchronous code, it's possible to use the runtime handle with `block_on()` as shown in the previous section. This, however, is probably not a common use case, and is something to be avoided.

For a more advanced discussion on this topic, please refer to the Tokio documentation at https://tokio.rs/tokio/topics/bridging.

8.8 When not to use async

Asynchronous programming is great for I/O heavy applications, such as network services. This could be an HTTP server, some other custom network service, or even a program which initiates many network requests (as opposed to responding to requests). Async *does* bring with it some complexity that you generally don't need to worry about with synchronous programming for the reasons outlined throughout this chapter.

It's reasonable to always use async as a general rule, but only in cases where concurrency is

required. Many programming tasks don't require concurrency and are best served by synchronous programming. An example of this could be a simple CLI tool which reads or writes to a file or standard I/O, or a simple HTTP client that makes a few sequential HTTP requests like cURL. If you have a cURL-like tool which needs to make thousands of concurrent HTTP requests, then by all means *do* use async.

It's worth noting that adding async after the fact is more difficult than building software with async support up front, so do think carefully about whether your use case doesn't require async. In terms of raw performance, there is practically no difference whether you use async or not for simple sequential and non-concurrent tasks, however Tokio introduces some *slight* overhead, which may be measurable, but unlikely to be significant for most purposes.

8.9 Tracing and debugging async code

For any sufficiently complex networked application, it's critical to instrument your code in order to measure its performance and debug problems. The Tokio project provides a tracing[41] crate for this purpose. The tracing crate supports the OpenTelemetry[42] standard which enables integration with a number of popular third party tracing and telemetry tools, but it can also emit traces as logs.

Enabling tracing with Tokio also unlocks tokio-console[43], which is a CLI tool similar to the `top` program you're likely familiar with from most UNIX systems. tokio-console allows you to analyze Tokio-based async Rust code *in real time*. Neat! While tokio-console is handy, in most environments you'd likely emit traces to logs or with OpenTelemetry, as tokio-console is ephemeral and mainly useful as a debug tool. You also cannot attach tokio-console to a program that wasn't compiled ahead of time for tokio-console.

Enabling tracing requires configuring a subscriber to which the traces are emitted. Additionally, to use tracing effectively, you need to instrument functions at the points where you want to measure them. This can be done easily with the `#[tracing::instrument]` macro. Traces can be emitted at different levels and with a number of options, which are well documented in the tracing docs at https://docs.rs/tracing/latest/tracing/index.html.

Let's write a small program to demonstrate tracing with tokio-console, which requires some setup and boilerplate. Our program will have 3 different sleep functions, each instrumented, and they will run forever concurrently in a loop:

```
use tokio::time::{sleep, Duration};

#[tracing::instrument]  ❶
async fn sleep_1s() {
    sleep(Duration::from_secs(1)).await;
}

#[tracing::instrument]
async fn sleep_2s() {
    sleep(Duration::from_secs(2)).await;
}

#[tracing::instrument]
async fn sleep_3s() {
    sleep(Duration::from_secs(3)).await;
}

#[tokio::main]
async fn main() {
    console_subscriber::init();  ❷

    loop {
        tokio::spawn(sleep_1s());  ❸
        tokio::spawn(sleep_2s());
        sleep_3s().await;  ❹
    }
}
```

❶ We'll use the instrument macro from the tracing crate to instrument our 3 sleep functions.

❷ We have to initialize the console subscriber in our main function to emit the traces.

❸ We'll fire and forget sleep 1 and sleep 2, then block on sleep 3.

❹ Here we block on sleep 3 until 3 seconds have elapsed, then repeat the process forever.

We also have to add the following to our dependencies, specifically to enable the tracing feature in Tokio:

```
[dependencies]
tokio = { version = "1", features = ["full", "tracing"] }  ❶
tracing =  "0.1"
console-subscriber = "0.1"
```

❶ The tracing feature in Tokio is *not* enabled by "full", it must be explicitly enabled.

We'll install tokio-console with `cargo install tokio-console`, after which we can compile and run our program. However, we need to compile with `RUSTFLAGS="--cfg tokio_unstable"` to enable unstable tracing features in Tokio for tokio-console. We'll do this by running the program directly from cargo with `RUSTFLAGS="--cfg tokio_unstable" cargo run`. With our program running, we can now run tokio-console, and it will produce output as seen below in figure 8.6:

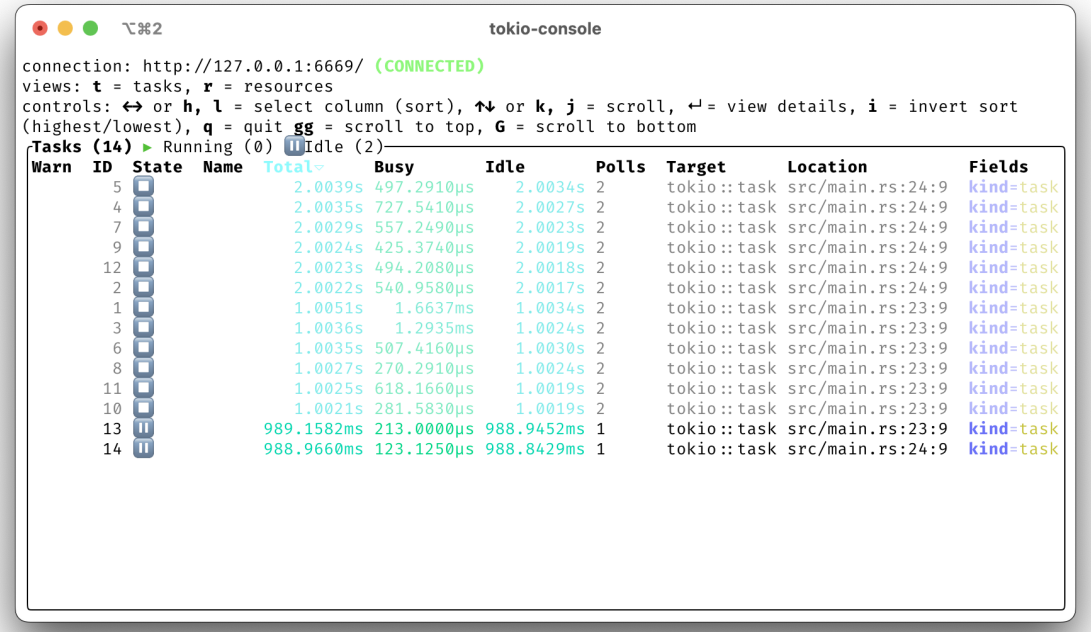

Figure 8.6 Running tasks as shown in tokio-console

In addition to monitoring tasks, we can monitor resources as shown in figure 8.7:

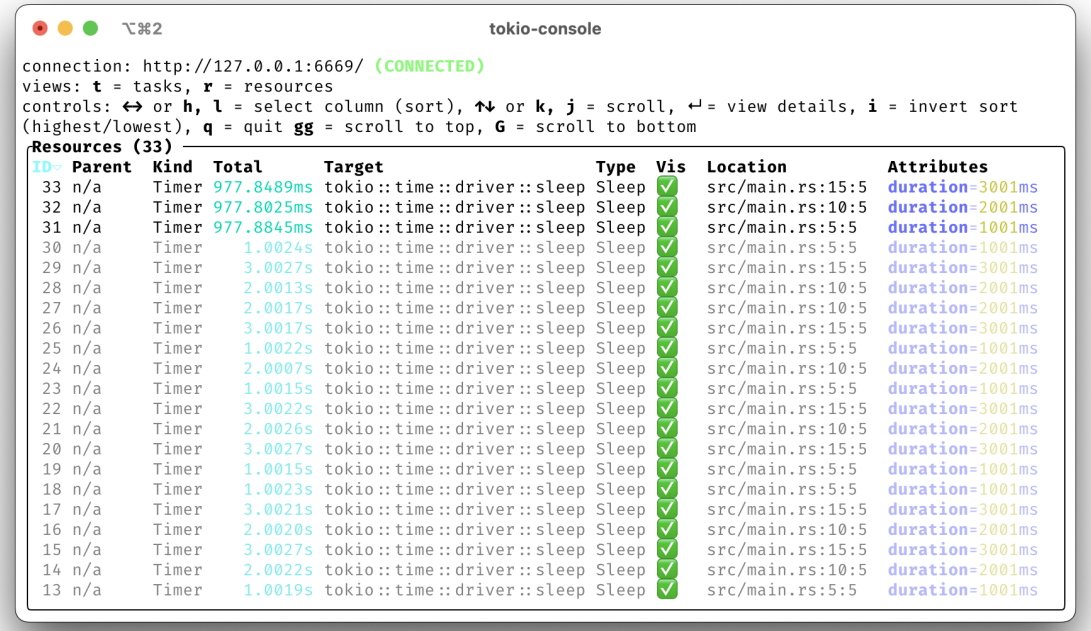

Figure 8.7 Resource usage as shown in tokio-console

We can also drill down into individual tasks, and even see a histogram of poll times as shown in

figure 8.8:

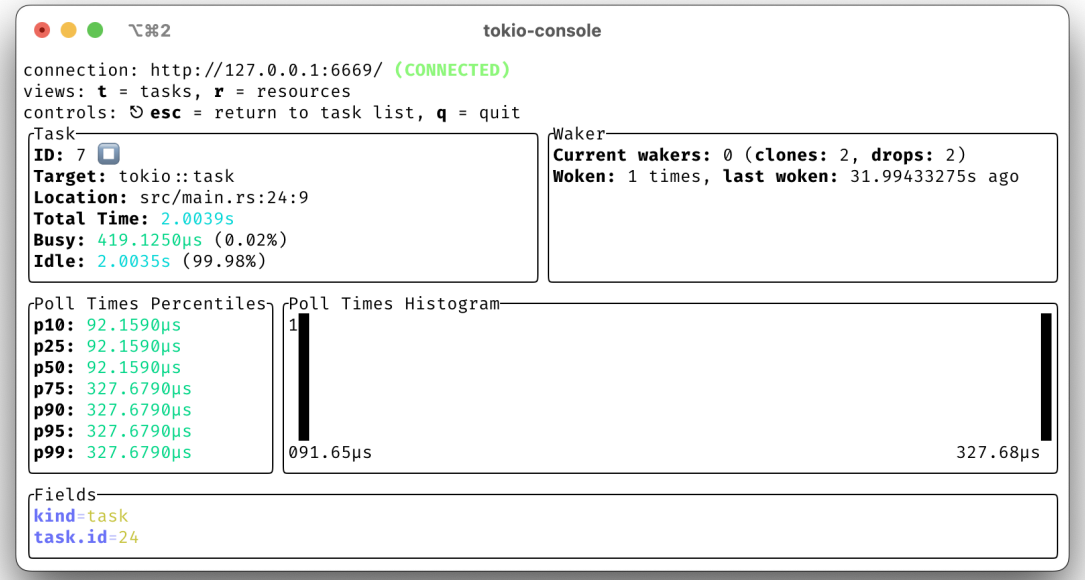

Figure 8.8 Individual task with poll time histogram

With tokio-console we can see the state of tasks in real time, a variety of metrics associated with each one, additional metadata we may have included, as well as source file locations. tokio-console let's us see both the tasks we've implemented, and the Tokio resources separately. All this data will also be made available in traces emitted to another sink such as a log file or an OpenTelemetry collector.

8.10 Dealing with async when testing

The last thing we'll discuss in this chapter is testing async code. When it comes to writing unit or integration tests for async code, there are 2 strategies:

- Creating and destroying the async runtime for each separate test.
- Reusing one or more async runtimes across separate tests.

For most cases it's preferable to create and destroy the runtime for each test, but there are exceptions to this rule where reusing a runtime is more sensible. In particular, it's reasonable to reuse the runtime if you have many (i.e., hundreds or thousands) of tests.

To reuse a runtime across tests, we can use the lazy_static crate which we discussed quite a bit in chapter 6. Rust's testing framework runs tests in parallel across threads, which must be handled correctly using `tokio::runtime::Handle` like we demonstrated earlier in this chapter.

For most cases, you can simply use the `#[tokio::test]` macro, which works exactly like

`#[test]` except that it's for async functions. The Tokio test macro takes care of setting up the test runtime for you, so you can write your async unit or integration test like you normally would any other test. To demonstrate, consider the following function which sleeps for 1 second:

```
async fn sleep_1s() {
    sleep(Duration::from_secs(1)).await;
}
```

We can write a test using the `#[tokio::test]` macro, which handles creating the runtime for us:

```
#[tokio::test]
async fn sleep_test() {
    let start_time = Instant::now();
    sleep(Duration::from_secs(1)).await;
    let end_time = Instant::now();
    let seconds = end_time
        .checked_duration_since(start_time)
        .unwrap()
        .as_secs();
    assert_eq!(seconds, 1);
}
```

The test above runs normally like any other test, except it's running within an async context. You may certainly manage the runtime yourself if you wish, but as mentioned already, you must be mindful of the fact that Rust's tests will run in parallel.

Lastly, Tokio provides the tokio_test crate, which can be enabled by adding the "test-util" feature (which is *not* enabled by the "full" feature flag). This includes some helper tools for mocking async tasks, as well as some convenience macros for use with Tokio. The tokio_test crate is documented at https://docs.rs/tokio-test/latest/tokio_test/.

8.11 Summary

- Rust provides multiple async runtime implementations, but most people should use Tokio for most purposes.
- Asynchronous programming requires special control flow, and our code must yield to the runtime's scheduler to allow it an opportunity to switch tasks. We can yield using `.await`, or by spawning futures directly with `tokio::task::spawn()`.
- Asynchronous code blocks (such as functions) are denoted with `async` keyword. Async code blocks always return futures.
- We can execute a future with the `.await` statement, but only within an async context (i.e., an async code block).
- Async blocks are lazy, and will not execute until we call `.await`, or are spawned explicitly. This differs from most other async implementations.
- Using `tokio::select! {}` or `tokio::join!()` allows us to introduce explicit concurrency.
- Spawning tasks with `tokio::task::spawn()` allows us to introduce concurrency *and* parallelism.
- If we want to perform blocking operations, we spawn them with `tokio::task::spawn_blocking()`.
- The tracing crate provides an easy way to instrument and emit telemetry to logs or OpenTelemetry collectors.
- We can use tokio-console with tracing to debug async programs.
- Tokio provides testing macros for unit and integration tests which provide the necessary testing runtime environment. The tokio_test crate, which can be enabled with the "test-util" feature flag on Tokio, provides mocking and assertion tools for use with Tokio.

11 Optimizations

This chapter covers

- Understanding Rust's zero cost abstractions
- Using vectors effectively
- Programming with SIMD in Rust
- Parallelization with Rayon
- Using Rust to accelerate other languages

In this final chapter, we'll discuss optimization strategies with Rust. Rust's zero cost abstractions allow you to confidently write Rust code without thinking too hard about performance. Rust delegates much of its machine code generation to LLVM, which has mature, robust, widely deployed, and well-tested code optimization. Code written in Rust will be fast and well-optimized without needing to spend time hand-tuning code.

There are, however, certain cases where you may need to dive deeper, and we'll discuss some of those cases and the tools you'll need in this chapter. We'll also discuss how you can use Rust to accelerate code from other languages, which is a fun way to introduce Rust into codebases without having to perform a complete re-write.

11.1 Zero cost abstractions

An important feature of Rust is its *zero cost abstractions*. In short, Rust's abstractions allow you to write high level code which produces optimized machine code with no additional runtime overhead. Rust's compiler takes care of figuring out how to get from high level Rust to low level machine code in the optimal way, without overhead. You can safely use Rust's abstractions without having to worry about whether it will create a performance trap.

The trade off to Rust's promise of zero cost abstractions is that some features from high level languages which you may have come to expect don't exist, and probably never will. Some of those features include virtual methods, reflection, function overloading, and optional function arguments. Rust provides alternatives to these, or ways to emulate their behaviour, but they're not baked into the language. If you want to introduce that overhead, you have to do it yourself (which of course makes it much easier to reason about).

We can compare Rust's abstractions to those of C++, for example, which *does* have runtime overhead. In the case of C++, the core *class* abstraction may contain virtual methods which require runtime lookup tables (called *vtables*). While this overhead is not usually significant, it *can* become significant in certain cases, such as calling virtual methods within a tight loop over many elements.

NOTE **Rust's trait objects do use vtables for method calls. Trait objects are a feature enabled by the `dyn Trait` syntax. For example, you can store a trait object with `Box<dyn MyTrait>`, where the item in the box must implement `MyTrait`, and the methods from `MyTrait` can be called using a vtable lookup.**

Reflection is another opaque abstraction which is used extensively in some languages to dispatch function calls or perform other operations at runtime. Reflection is often used in Java, for example, to handle a variety of problems, but it also tends to generate a significant number of difficult to debug runtime errors. Reflection offers a bit of convenience for the programmer with the trade off of flakier code.

Rust's zero cost abstractions are based on compile time optimizations. Within that framework, Rust can optimize out unused code or values as needed. Rust's abstractions can also be deeply nested, and the compiler can (in most cases) perform optimizations all the way down the abstraction chain. When we talk about "zero cost abstractions" in Rust, what we *really* mean is "zero cost abstractions once all optimizations have been performed".

When we want to build a production binary, or benchmark our code, we must enable compiler optimizations by compiling our code in release mode by enabling the `--release` flag with cargo, as the default compilation mode is debug. If you forget to enable release mode, you may experience unexpected performance penalties, one of which we'll demonstrate in the next section.

11.2 Vectors

Vectors are the core collection abstraction in Rust. As I've mentioned throughout this book, you should use `Vec` in most cases where you require a collection of elements. Because you'll be using `Vec` so often in Rust, it's important to understand a few details about its implementation and how it can affect the performance of your code. Additionally, you should understand when it makes sense to use something other than a `Vec`.

The first thing to understand about `Vec` is how memory is allocated. We have discussed this a fair bit already in chapters 4 and 5, but we'll go into a little more detail here. `Vec` allocates memory in contiguous blocks, with a configurable chunk size based on capacity. It allocates memory lazily by delaying allocation until it's necessary, and always in contiguous blocks.

11.2.1 Vector memory allocation

The first thing you should understand about the way `Vec` allocates memory is how it determines capacity size. By default, an empty `Vec` has a capacity of 0 and thus no memory is allocated. It's not until data is added that memory allocation occurs. When the capacity limit is reached, `Vec` will double the capacity (i.e., capacity increases exponentially).

We can see how `Vec` adds capacity by running a small test:

```
let mut empty_vec = Vec::<i32>::new();
(0..10).for_each(|v| {
    println!(
        "empty_vec has {} elements with capacity {}",
        empty_vec.len(),
        empty_vec.capacity()
    );
    empty_vec.push(v)
});
```

Note that capacity is measured in number of elements, not number of bytes. The number of bytes required for the vector is the capacity multiplied by the size of each element. When we run the code above, it generates the following output:

```
empty_vec has 0 elements with capacity 0
empty_vec has 1 elements with capacity 4   ❶
empty_vec has 2 elements with capacity 4
empty_vec has 3 elements with capacity 4
empty_vec has 4 elements with capacity 4
empty_vec has 5 elements with capacity 8   ❷
empty_vec has 6 elements with capacity 8
empty_vec has 7 elements with capacity 8
empty_vec has 8 elements with capacity 8
empty_vec has 9 elements with capacity 16  ❸
```

❶ Capacity is increased from 0 to 4.

❷ Capacity is increased from 4 to 8.

❸ Capacity is increased from 8 to 16.

We can examine the source code from the Rust standard library to see the algorithm itself which is part of `RawVec`, the internal data structure used by `Vec` in 11.1:

Listing 11.1 Listing of `Vec::grow_amortized()` from Rust standard library

```
// This method is usually instantiated many times. So we want it to be as
// small as possible, to improve compile times. But we also want as much of
// its contents to be statically computable as possible, to make the
// generated code run faster. Therefore, this method is carefully written
// so that all of the code that depends on `T` is within it, while as much
// of the code that doesn't depend on `T` as possible is in functions that
// are non-generic over `T`.
fn grow_amortized(&mut self, len: usize, additional: usize) ->
 ➥ Result<(), TryReserveError> {
    // This is ensured by the calling contexts.
    debug_assert!(additional > 0);

    if mem::size_of::<T>() == 0 {
        // Since we return a capacity of `usize::MAX` when `elem_size` is
        // 0, getting to here necessarily means the `RawVec` is overfull.
        return Err(CapacityOverflow.into());
    }

    // Nothing we can really do about these checks, sadly.
    let required_cap = len.checked_add(additional).ok_or(CapacityOverflow)?;

    // This guarantees exponential growth. The doubling cannot overflow
    // because `cap <= isize::MAX` and the type of `cap` is `usize`.
    let cap = cmp::max(self.cap * 2, required_cap);   ❶
    let cap = cmp::max(Self::MIN_NON_ZERO_CAP, cap);   ❷

    let new_layout = Layout::array::<T>(cap);

    // `finish_grow` is non-generic over `T`.
    let ptr = finish_grow(new_layout, self.current_memory(), &mut self.alloc)?;
    self.set_ptr_and_cap(ptr, cap);
    Ok(())
}
```

❶ Here the capacity (`self.cap`) is doubled.

❷ `Self::MIN_NON_ZERO_CAP` varies depending on the size of the elements, but it can be either 8, 4, or 1.

What can we do with this information? There are 2 main takeaways:

- The lazy allocation by `Vec` can be inefficient if you're adding many elements a few at a time.
- For large vectors, the capacity will be up to twice the number of elements in the array.

The first issue (lazy allocation) can be problematic in cases where you are frequently creating new vectors and pushing data into them. Reallocations are costly because they can involve shuffling memory around. Reallocations with a small number of elements aren't as costly

because your machine likely has lots of space in memory for small contiguous regions, but as the structure grows it can become harder and harder to find available contiguous regions (thus, more memory shuffling required).

The second problem with large vectors can be mitigated either by using a different structure (such as a linked list), or by keeping the capacity trimmed with the `Vec::shrink_to_fit()` method. It's also worth noting that vectors can be large in two different dimensions: a large number of small elements, or a small number of large elements. For the latter case (a few large elements) a linked list or storing elements within a `Box` will provide relief from memory pressure.

11.2.2 Vector iterators

Another important thing to consider when discussing `Vec` performance is iterating over elements. There are 2 ways to loop over a `Vec`, either using `iter()` or `into_iter()`. The `iter()` iterator allows us to iterate over elements with references, whereas `into_iter()` consumes `self`.

Let's look at 11.2 to analyze `Vec` iterators:

Listing 11.2 Demonstrating `Vec` iterator performance

```
let big_vec = vec![0; 10_000_000];
let now = Instant::now();
for i in big_vec {
    if i < 0 {
        println!("this never prints");
    }
}
println!("First loop took {}s", now.elapsed().as_secs_f32());

let big_vec = vec![0; 10_000_000];
let now = Instant::now();
big_vec.iter().for_each(|i| {
    if *i < 0 {
        println!("this never prints");
    }
});
println!("Second loop took {}s", now.elapsed().as_secs_f32());
```

Above we have some large vectors which we're going to iterate over with a no-op block of code to prevent the compiler from optimizing it out. We'll test the code with `cargo run --release` because we want to enable all compiler optimizations. Now, if I run the code above it will produce the following output:

```
First loop took 0.007614s
Second loop took 0.00410025s
```

Woah! What happened there? Why does the `for` loop run nearly twice as slow?

The answer involves a bit of sugar syntax: the `for` loop expression roughly translates into using the `into_iter()` method to obtain an iterator, and looping over the iterator until hitting the end

(the full expression is documented at https://doc.rust-lang.org/reference/expressions/loop-expr.html#iterator-loops). `into_iter()` takes `self` by default, meaning it consumes the original vector and (in some cases) may even require allocating an entirely new structure.

The `for_loop()` method provided by the core iterator trait in Rust, however, is highly optimized for this purpose, which gives us a slight performance gain. Additionally, `iter()` takes `&self` and iterates over references to elements in the vector, which can be further optimized by the compiler.

To verify this, we can update our code to use `into_iter()` instead of `iter()`. Let's try by adding a third loop:

```
let big_vec = vec![0; 10_000_000];
let now = Instant::now();
big_vec.into_iter().for_each(|i| {
    if i < 0 {
        println!("this never prints");
    }
});
println!("Third loop took {}s", now.elapsed().as_secs_f32());
```

Then we can run the code again in release mode to produce and following output:

```
First loop took 0.011229166s
Second loop took 0.005076166s
Third loop took 0.008608s
```

That's much closer, however it seems that using iterators directly instead of `for` loop expressions is even slightly faster. Out of curiosity, what happens if we run the same test in debug mode? Let's try and see what it produces:

```
First loop took 0.074964s
Second loop took 0.14158678s
Third loop took 0.07878621s
```

Wow, those results are drastically different! What's particularly interesting is that in debug mode `for` loops are slightly faster. This is likely because of the additional overhead imposed by enabling debugging symbols and disabling compiler optimizations. The lesson here is that benchmarking performance in debug mode will lead to strange results.

Vectors include quite a few other built in optimizations, but the ones you'll generally need to concern yourself with are memory allocation and iterators. We can decrease the amount of memory allocations required by pre-allocating memory with `Vec::with_capacity()`, and we can avoid confusing performance issues by using iterators directly rather than with `for` loop expressions.

11.2.3 Fast copies with *Vec* and slices

Let's discuss one more optimization with vectors, which has to do with copying memory. Rust has a fast-path optimization for vectors and slices where it can perform a faster copy of everything within a `Vec` under certain circumstances. The optimization lives inside the `Vec::copy_from_slice()` method, for which the key parts of the implementation are shown in 11.3:

Listing 11.3 Partial listing of `copy_from_slice()` from Rust standard library

```
pub fn copy_from_slice(&mut self, src: &[T])
where
    T: Copy,
{
    // ... snip ...
    unsafe {
        ptr::copy_nonoverlapping(src.as_ptr(), self.as_mut_ptr(), self.len());
    }
}
```

In the partial listing above lifted from the Rust standard library, you'll notice 2 important things: the trait bound `Copy`, and the unsafe call to `ptr::copy_nonoverlapping`. In other words, if you use a `Vec` and you want to copy items between two vectors, provided those items implement `Copy` you can take the fast path. We can run a quick benchmark to see the difference in 11.4:

Listing 11.4 Benchmarking `ptr::copy_nonoverlapping()`

```
let big_vec_source = vec![0; 10_000_000];
let mut big_vec_target = Vec::<i32>::with_capacity(10_000_000);  ❶
let now = Instant::now();
big_vec_source
    .into_iter()
    .for_each(|i| big_vec_target.push(i));
println!("Naive copy took {}s", now.elapsed().as_secs_f32());

let big_vec_source = vec![0; 10_000_000];
let mut big_vec_target = vec![0; 10_000_000];
let now = Instant::now();
big_vec_target.copy_from_slice(&big_vec_source);
println!("Fast copy took {}s", now.elapsed().as_secs_f32());
```

❶ We initialize the target `Vec` with pre-allocated memory of the correct size.

Running the code above in release mode produces the following result:

```
Naive copy took 0.024926165s
Fast copy took 0.003599458s
```

In other words, using `Vec::copy_from_slice()` gives us about an 8x speedup in copying data directly from one vector to another. This optimization also exists for the slice (`&mut [T]`) and array (`mut [T]`) types.

11.3 SIMD

There may come a point in your life as a developer where you need to use *SIMD*, which stands for *single instruction, multiple data*. SIMD is a hardware feature of many modern microprocessors that allows performing operations on sets of data simultaneously with a single instruction. The most common use case for this is either to optimize code for a particular processor, or to guarantee consistent timing of operations (such as to avoid timing attacks in cryptographic applications).

SIMD is platform dependent: different CPUs have different SIMD features available, but almost all modern CPUs have some SIMD features. The most commonly used SIMD instruction sets are MMX, SSE, and AVX on Intel devices, and Neon on ARM devices.

In the past, if you needed to use SIMD you'd need to write inline assembly yourself. Thankfully today modern compilers provide an interface for using SIMD without needing to write assembly directly. These functions standardize some of the shared behaviour among the different SIMD implementations in a somewhat portable way. The advantage of using portable SIMD is that we don't need to worry about instruction set details for any particular platform, with the tradeoff that we only have access to the common denominator features. It's still possible to write inline assembly if you choose, but I'm going to focus on *portable* SIMD.

One convenient feature of portable SIMD is that the compiler can automatically generate substitute non-SIMD code for cases where features are not available at the hardware level.

The Rust standard library provides the `std::simd` module, which is currently a nightly-only experimental API. Documentation for the portable SIMD API can be found at https://doc.rust-lang.org/std/simd/struct.Simd.html.

To illustrate, we can write a benchmark to compare the speed of some math operations on 64 element arrays with and without SIMD as shown in 11.5:

Listing 11.5 Multiplying vectors with SIMD versus iterators

```
#![feature(portable_simd, array_zip)]   ❶

fn initialize() -> ([u64; 64], [u64; 64]) {
    let mut a = [0u64; 64];
    let mut b = [0u64; 64];
    (0..64).for_each(|n| {
        a[n] = u64::try_from(n).unwrap();
        b[n] = u64::try_from(n + 1).unwrap();
    });
    (a, b)
}

fn main() {
    use std::simd::Simd;
    use std::time::Instant;

    let (mut a, b) = initialize();   ❷

    // perform some calculations using normal math
    let now = Instant::now();
    for _ in 0..100_000 {
        let c = a.zip(b).map(|(l, r)| l * r);
        let d = a.zip(c).map(|(l, r)| l + r);
        let e = c.zip(d).map(|(l, r)| l * r);
        a = e.zip(d).map(|(l, r)| l ^ r);   ❸
    }
    println!("Without SIMD took {}s", now.elapsed().as_secs_f32());

    let (a_vec, b_vec) = initialize();   ❹

    let mut a_vec = Simd::from(a_vec);   ❺
    let b_vec = Simd::from(b_vec);

    // perform the same calculations with SIMD
    let now = Instant::now();
    for _ in 0..100_000 {
        let c_vec = a_vec * b_vec;
        let d_vec = a_vec + c_vec;
        let e_vec = c_vec * d_vec;
        a_vec = e_vec ^ d_vec;   ❻
    }
    println!("With SIMD took {}s", now.elapsed().as_secs_f32());

    // check the final result is the same in both
    assert_eq!(&a, a_vec.as_array());   ❼
}
```

❶ Enables experimental features for this crate.

❷ Initializes our 64 element arrays.

❸ Stores the result back into `a`.

❹ Initialize with the same values again.

❺ Convert our arrays into SIMD vectors.

❻ Stores the result back into `a_vec`.

❼ Finally, check that `a` and `a_vec` have the same result.

Running the code above will produce the following output:

```
Without SIMD took 0.07886646s
With SIMD took 0.002505291s
```

Wow! We got a nearly 40x speedup by using SIMD. Additionally, the SIMD code provides consistent timing, which is important for applications that are timing dependent such as cryptography.

11.4 Parallelization with rayon

For problems where performance can be improved with the use of parallelization (i.e., parallel programming with threads), the rayon crate is your best place to start. While the Rust language certainly provides threading features as part of its core library, these features are somewhat primitive, and most of the time it's better to write code based on higher-level APIs.

Rayon provides two ways to interact with data in parallel: a parallel iterator implementation, and some helpers for creating lightweight tasks based on threads. We're going to focus mainly on rayon's iterator, because that is the most useful part of the library.

The typical use case for rayon is one where you have a significant number of tasks, each of which are relatively long running or compute intense. If you have a small number of tasks, or your tasks are not very compute intense, introducing parallelization is likely to actually *decrease* performance. Generally speaking, parallelization has diminishing returns as the number of threads increases, due to synchronization and data starvation issues that can arise when moving data between threads (as shown in 11.1).

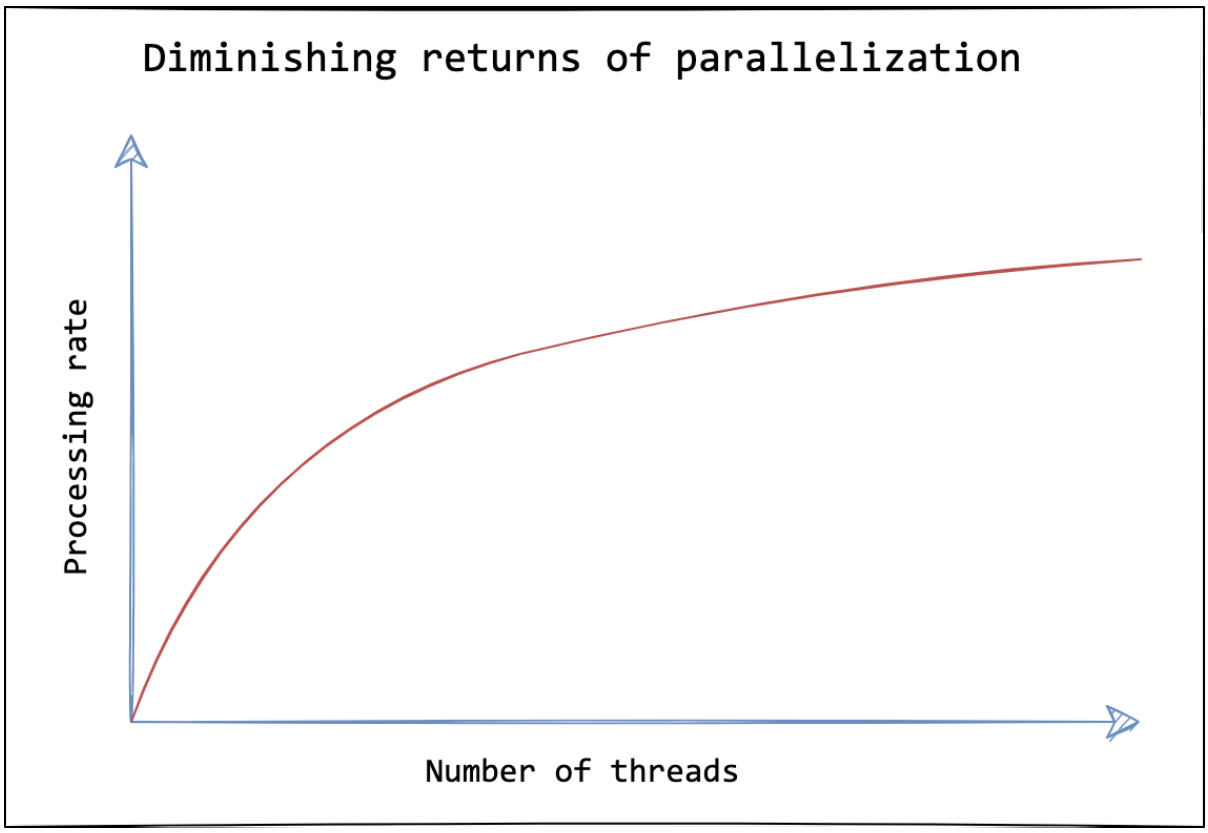

Figure 11.1 Diminishing returns with increased parallelization.

One handy feature of rayon's iterators is that they're mostly compatible with the core `Iterator` trait, which makes it very easy to quickly benchmark code with or without parallelization. Let's demonstrate by creating two different tests: one which will be slower with rayon, and one which will be faster with rayon.

First, let's write a test that's faster *without* rayon:

```
let start = Instant::now();
let sum = data
    .iter()
    .map(|n| n.wrapping_mul(*n))
    .reduce(|a: i64, b: i64| a.wrapping_add(b));  ❶
let finish = Instant::now() - start;
println!(
    "Summing squares without rayon took {}s",
    finish.as_secs_f64()
);

let start = Instant::now();
let sum = data
    .par_iter()
    .map(|n| n.wrapping_mul(*n))
    .reduce(|| 0, |a: i64, b: i64| a.wrapping_add(b));  ❷
let finish = Instant::now() - start;
println!("Summing squares with rayon took {}s", finish.as_secs_f64());
```

❶ We use `reduce()` instead of `sum()` because `sum()` performs addition without handling overflow.

❷ Notice the slight difference in signature of `reduce()` with rayon, which requires an identity value which may be inserted to create opportunities for parallelization.

In the code above, we've generated an array filled with random integer values. Next, we square each value and then calculate the sum across the whole set. This is a classic map/reduce example. If we run the code above, we get the following result:

```
Summing squares without rayon took 0.000028875s
Summing squares with rayon took 0.000688583s
```

This test *with* rayon takes 23x longer than without! This is clearly not a good candidate for parallelization.

Let's construct another test, which will use a regular expression to scan a long string for a word. We'll randomly generate some very large strings before we run the search. The code looks like this:

```
let re = Regex::new(r"catdog").unwrap();  ❶

let start = Instant::now();
let matches: Vec<_> = data.iter().filter(|s| re.is_match(s)).collect();
let finish = Instant::now() - start;
println!("Regex took {}s", finish.as_secs_f64());

let start = Instant::now();
let matches: Vec<_> =
    data.par_iter().filter(|s| re.is_match(s)).collect();
let finish = Instant::now() - start;
println!("Regex with rayon took {}s", finish.as_secs_f64());
```

❶ `Regex` is both `Send` and `Sync`, we can use it in a parallel filter with rayon.

Running the code above, we get the following result:

```
Regex took 0.043573333s
Regex with rayon took 0.006173s
```

In this case, parallelization with rayon gives us a 7x speedup. Scanning strings is one case where we might see a significant boost in performance across sufficiently large data sets.

Other notable features of rayon include its parallel sorting implementation, which allows you to sort slices in parallel. For larger datasets, this can provide a decent performance boost. Rayon's `join()` also provides a work-stealing implementation which executes tasks in parallel when idle worker threads are available, but you should use parallel iterators instead when possible.

For more details on rayon, consult the documentation at https://docs.rs/rayon/latest/rayon/index.html.

11.5 Using Rust to accelerate other languages

The last thing we'll discuss in this chapter is one of the coolest application of Rust: calling Rust code from other languages to perform operations that are either safety critical, or compute intensive. This is a common pattern with C and C++, too: many language runtimes implement performance critical features in C or C++. With Rust, however, you have the added bonus of Rust's safety features. This, in fact, was one of the major motivating factors behind the adoption of Rust by several organizations, such as Mozilla with their plans to improve the security and performance of the Firefox browser.

An example of how you might use Rust in this case would be to parse or validate data from external sources, such as a web server receiving untrusted data from the internet. Many security vulnerabilities are discovered by feeding random data into public interfaces and seeing what happens, and often code contains mistakes (such as reading past the end of a buffer) which aren't possible in Rust.

Most programming languages and runtimes provide some form of FFI (foreign function interface) bindings, which we demonstrated in chapter 4. However, for many popular languages there are higher-level bindings and tooling available which can make integrating Rust much easier than dealing with FFI, and some also help with packaging native binaries.

Table 11.1 Rust bindings and tooling for integrating Rust into other languages

Language	Name	Description	URL	GitHub stars [44]
Python	PyO3	Rust bindings for Python, with tools for making native Python packages with Rust	https://pyo3.rs	6,340
Python	Milksnake	setuptools extension for including binaries in Python packages, including Rust	https://github.com/getsentry/milksnake	737
Ruby	ruru	Library for building native Ruby extensions with Rust	https://github.com/d-unseductable/ruru	804
Ruby	rutie	Bindings between Ruby and Rust, which enables integrating Rust with Ruby or Ruby with Rust	https://github.com/danielpclark/rutie	665
Elixir and Erlang	rustler	Library for creating safe bindings to Rust for Elixir and Erlang	https://github.com/rusterlium/rustler	3,316
JavaScript and TypeScript on Node.js	Neon	Rust bindings for creating native Node.js modules with Rust	https://neon-bindings.com	6,746
Java	jni-rs	Native Rust bindings for Java	https://github.com/jni-rs/jni-rs	685
Rust	bindgen	Generates Rust FFI bindings from native Rust	https://github.com/rust-lang/rust-bindgen	2,871

11.6 Where to go from here

Congratulations on making it to the end of *Code Like a Pro in Rust*. Let's take a moment to reflect on what we've discussed in this book, and more importantly, where to go from here to learn more.

In chapters 1 through 3, we focused on tooling, project structure, and the basic skills you need to work effectively with Rust. Chapters 4 & 5 covered data structures and Rust's memory model. Chapters 6 & 7 focused on Rust's testing features and how to get the most out of them. In chapters 8, 9, and 10 we dove deep into what it takes to write Rustaceous code, including the implementation of higher level design patterns, which put our skills into practice. Chapter 11 introduced us to async Rust, and this final chapter focused on optimization opportunities for Rust code.

At this point in the book, you may want to take some time to go back and revisit previous sections, especially if you found the content dense or hard to grok. It's often good to give your brain a rest, then return to problems once you've had time to digest new information. For further reading on Rust, check out Tim McNamara's *Rust in Action* from Manning Publications.

Rust and its ecosystem is forward-looking, and always evolving. The Rust language, while already quite mature, is actively developed and continuously moving forward. As such, I will close out the final chapter of this book by leaving you with some resources on where to go from here to learn about new Rust features, changes, proposals, and how to get more involved with the Rust community.

The majority of Rust language and tool development is hosted on GitHub, under the *rust-lang* project at https://github.com/rust-lang. Additionally, the following resources are a great place to dive deeper on the Rust language:

- https://github.com/rust-lang/rust/blob/master/RELEASES.md: release notes for each Rust version
- https://rust-lang.github.io/rfcs: Rust RFCs ("request for comments"), proposed Rust features and their current status
- https://doc.rust-lang.org/reference: the official Rust language reference
- https://users.rust-lang.org: the official Rust language users forum, where you can discuss Rust with other like-minded people

11.7 Summary

- Rust's zero cost abstractions allow you to write fast code without needing to worry about overhead, but code must be compiled in release mode to take advantage of these optimizations.
- Vectors, Rust's core sequence data structure, should be pre-allocated with the capacity needed, *if* you know the required capacity ahead of time.
- When copying data between structures, the `copy_from_slice()` method provides a fast path for moving data between slices.
- Rust's experimental portable SIMD feature allows you to easily build code using SIMD. You don't need to deal directly with assembly code, compiler intrinsics, or worry about which instructions are available.
- Code can be easily parallelized using the rayon crate, which builds on top of Rust's iterator pattern. Building parallelized code in Rust is as easy as using iterators.
- We can introduce Rust into other languages by swapping out individual components for Rust equivalents, getting the benefit of Rust's safety and performance features without needing to completely rewrite our applications.

Notes

1. https://insights.stackoverflow.com/survey/2021#most-loved-dreaded-and-wanted-language-love-dread
2. As of Sept 24, 2022
3. https://semver.org
4. https://crates.io/crates/semver
5. https://github.com/features/actions
6. https://github.com/brndnmtthws/dryoc/blob/main/.github/workflows/build-and-test.yml
7. https://github.com/brndnmtthws/dryoc/blob/main/.github/workflows/publish.yml
8. https://crates.io/crates/rand
9. https://rocket.rs/
10. https://crates.io/crates/protoc-rust
11. https://www.qemu.org
12. https://crates.io/crates/ruduino
13. https://llvm.org
14. https://doc.rust-lang.org/core/alloc/trait.GlobalAlloc.html
15. https://microsoft.github.io/language-server-protocol
16. https://plugins.jetbrains.com/plugin/8182-rust/docs/rust-faq.html#use-rls-racer-rustanalyzer
17. https://llvm.org/docs/LibFuzzer.html
18. http://jemalloc.net/
19. https://github.com/google/tcmalloc

20. https://crates.io/crates/parameterized

21. https://crates.io/crates/test-case

22. https://crates.io/crates/rstest

23. https://crates.io/crates/assert2

24. https://lib.rs/crates/proptest

25. https://en.wikipedia.org/wiki/QuickCheck

26. https://crates.io/crates/lazy_static

27. https://en.wikipedia.org/wiki/Fizz_buzz

28. https://crates.io/crates/cargo-tarpaulin

29. https://about.codecov.io/

30. https://coveralls.io/

31. https://en.wikipedia.org/wiki/Quicksort

32. https://curl.se/

33. https://github.com/httpie/httpie

34. https://crates.io/crates/assert_cmd

35. https://crates.io/crates/rexpect

36. https://crates.io/crates/assert_fs

37. https://llvm.org/docs/LibFuzzer.html

38. https://crates.io/crates/arbitrary

39. Number of downloads for each crate, as of July 21, 2022.

40. https://crates.io/crates/rocket

41. https://crates.io/crates/tracing

42. https://opentelemetry.io/

43. https://github.com/tokio-rs/console

44. GitHub star count as of 11 Aug, 2022