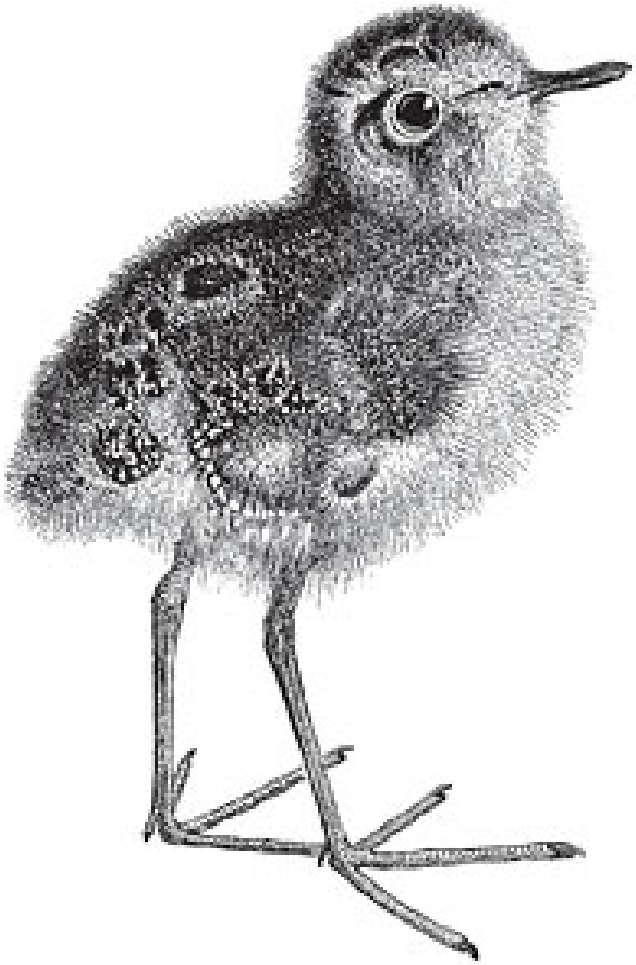
O'REILLY®

Systems Programming with Rust

A Project-Based Primer



Early Release

RAW & UNEDITED

Ken Youens-Clark

Systems Programming with Rust

A Project-Based Primer

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

Ken Youens-Clark

Systems Programming with Rust

by Charles Kenneth Youens-Clark

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://oreilly.com*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Acquisitions Editor: Suzanne McQuade

Development Editor: Corbin CollinsE

Production Editor: Caitlin Ghegan

Copyeditor: TO COME

Proofreader: TO COME

Indexer: TO COME

Interior Designer: David Futato

Cover Designer: TO COME

Illustrator: Kate Dullea

February 2022: First Edition

Revision History for the Early Release:

- 2021-06-18: First Release

- 2021-08-23: Second Release

See http://oreilly.com/catalog/errata.csp?isbn=9781098109431 for release details.

978-1-098-10941-7

[???]

Preface

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the Preface of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the author at kyclark@gmail.com.

I already know the ending it's the part that makes your face implode

—They Might Be Giants

I remember years back when this new language called "JavaScript" came out. It sounded interesting, so I bought a big, thick reference book on the language and read it cover to cover. When I finished, I couldn't write JavaScript to save my life. My problem was that I hadn't written any programs in the language. I've since learned better how to learn a language, which is perhaps the most valuable skill you can develop as a programmer. When I want to learn a language now, I start by rewriting things I already know like Tic-Tac-Toe.

Rust is reputed to have a fairly steep learning curve, but I'm convinced you can learn it much more quickly by writing many small programs you already know. This book is about two things: systems programming and Rust. Writing Rust versions of basic systems tools like `head` and `cal` will reveal patterns that you'll be able to use when you write your own programs—patterns like validating parameters, reading and writing file handles, parsing

text, and using regular expressions.

What Is Rust (And Why Is Everybody Talkin' About It)?

Rust is a programming language created by Graydon Hoare while working at Mozilla Research. He first started working on it around 2006 as a personal project, and by 2010 Mozilla had sponsored and announced the project. While the language is relatively new, it has quickly earned a passionate following of programmers who claim to even *love* using it.

Figure P-1. Here is a logo I made from an old Rush logo. Being a kid playing the drums in the 1980s, you better believe I listened to a lot of Rush. Anyway, Rust is cool, and this logo proves it.

Often Rust is described as a systems programming language that has been designed for performance and safety. It has a syntax that resembles the C language, so you'll find things like `for` loops, semicolon-terminated statements, and curly braces denoting block structures.

Rust is also a *statically typed* language, meaning that a variable can never change its type, like from a number to a string. If you're coming from languages like C/C++ or Java, this will be familiar because those languages are also statically typed. You don't always have to declare a variable's type in Rust because the compiler can often figure it out from the context, but the variable is still never allowed to change its type.

If, like me, you are coming from a *dynamically typed* language like Perl, JavaScript, or Python, this could feel rather new. In those languages, a variable can change its type at any point in the program, like from a string to a file handle. The language may also silently treat a variable of one type like another; for instance, both Perl and Python will evaluate strings, numbers, lists, dictionaries, and more in a Boolean context which can lead to some surprising results. Interestingly, several dynamically typed languages including Python, Perl 6/Raku, and Julia have introduced type hints or *gradual typing* systems where a variable can be basically untyped (`Any`) or some specific type like an integer or string.

Although Rust bears a strong resemblance to imperative, C-style languages, it has borrowed many exciting concepts from other languages and programming paradigms. Rust is not an object-oriented (OO) language like Java as there are no classes or inheritance in Rust. Instead Rust uses a `struct` (structure) to represent complex data types and traits to describe how types can behave. These structures can have methods, can mutate the internal state of the data, and might even be called *objects* in the documentation, but they are not objects in the formal sense of the word.

Rust has borrowed many exciting ideas from purely functional languages like Haskell. For instance, variables are *immutable* by default, meaning they can't be changed. Like many languages, functions are *first-class* values, which means they can be passed as arguments. Most exciting to my mind is Rust's

use of *enumerated* and *sum* types, also called *algebraic data types* (ADTs), which allow you to represent, for instance, that a function can return a `Result` which can be either an `Ok` containing some value or an `Err` containing some other kind of value. Any code that deals with these values must handle all possibilities, so you're never at risk of forgetting to handle an error that could unexpectedly crash your program.

Who Should Read This Book

If you're reading this, I imagine you already know at least one programming language. Maybe your background is in statically typed languages like C++ or Java, or maybe like me, you're coming from the world of dynamically typed languages. I've spent the bulk of my career using Perl and Python for web development and systems tasks, but I've been very interested in learning everything from Prolog and Haskell to JavaScript and Lisp. No matter your background, I hope to show you that Rust is a really fun language.

Rust requires learning some pretty low-level stuff about memory and types, so I can't imagine it would be a great first language. I'm going to offload much of the nitty-gritty to reference books like *Programming Rust* (Blandy, Orendorff, and Tindall; O'Reilly, 2021) and *The Rust Programming Language* (Klabnik and Nichols; No Starch Press, 2019). I highly recommend that you read one or both of those along with this book to dig deeper into the language itself.

This book will focus on how to write practical programs, starting from scratch and working step-by-step to add features, build your program, work through error messages, and test your logic. You should read this book if you want to learn how to write complete programs that solve common systems problems. Although I'll be showing why Rust is particularly well-suited for this, I think you should also try writing these programs in other languages you know so that you can contrast what makes Rust better or worse, easier or harder.

Lastly, you should also read this book if you care about testing. I'm an advocate for *test-driven development* where I write tests first and then try to

write code that passes those tests. My experience teaching has convinced me that this is an effective way to think through problems. It's my opinion that testing generally produces better code. If you aren't currently using tests, I hope that you'll see enough examples of their utility here to adopt this practice.

Why You Should Learn Rust

There are many reasons why you might be interested to learn Rust. Personally, Rust's type system was very attractive to me. The more I used statically typed languages, the more I realized that dynamically typed languages force me, the programmer, to write correct programs and tests. The more I used Rust, the more I found that the compiler was my dance partner, not my enemy. Granted, it's a dance partner who will tell you every time you step on their toes or miss a cue, but that eventually makes you a better dancer, which is the goal after all.

One thing I like about Rust is how easily I can share a program I've written with someone who is not a developer. If I write a Python program for a workmate, I must give them the Python source code to run. This means that I must also ensure the user has installed the right version of Python and all the required modules to execute my code. In contrast, Rust programs are compiled directly into a machine-executable file. I can write and debug a program on my machine, build an executable for the architecture it needs to run on, and give my workmates a copy of the program. Assuming they have the correct architecture, they would not need to install Rust and could run the program directly.

The ease and size of distribution for Rust programs also applies to the containers I create. For instance, a Docker container with the Python runtime may require several hundred MB. In contrast, I can build a bare-bones Linux VM with a Rust binary that may only be tens of MB in size. Unless I really need some particular features of Python such as machine learning or natural language processing modules, I would prefer to write in Rust and have smaller, leaner containers.

Mostly what I've found with Rust is that I'm extremely productive. For a long time, I've had a school-boy crush on languages like Lisp and Haskell, but I've never successfully written programs in those languages that do anything useful. To be clear, I generally write command-line programs that need to accept some arguments, read and write data to files or a database, fetch data from the internet, and so forth. Languages like Haskell would leave me overwhelmed with libraries that have inscrutable documentation and produce unintelligible error messages. Conversely, I can easily find many useful and well-documented Rust crates—which is what libraries are called in Rust—on crates.io, and working with the compiler feels like pair programming with an extremely smart coworker who finds errors and often suggests exactly how to fix them.

The Coding Challenges

This is a book of coding challenges for you to write. I don't want you to passively read this book on the bus to work and put it away. For each chapter, I want you to write a program that will pass the test suite I've written for you. You will learn the most by writing your own solutions, but I believe that even typing the source code I present will prove beneficial.

The problems I've selected hail mostly from the Unix command line core utils because I expect these will already be quite familiar to the reader. For instance, I assume you've used `head` and `tail` to look at the first or last few lines of a file, but have you ever written your own versions of these programs? There seems to be a desire in Rust to rewrite existing programs, especially ones originally written in C or C++, so that's another part of my motivation. You'll be able to find many other examples of these programs written by Rustaceans on the web, so you can contrast your code with my solutions and the others you'll find. Beyond that, these are fairly simple programs that lend themselves to teaching a few skills in each program. I've sequenced them so that they build upon each other, so it's probably best if you work through all the chapters in order.

One reason I've chosen many of these programs is that they provide a sort of

Ground Truth. While there are many flavors of Unix and many implementations of these programs, they usually all work the same and produce the same results. I use macOS for my development, which means I'm running mostly the BSD (Berkeley Standard Distribution) or GNU (GNU's Not Unix) variants of these programs. Generally speaking, the BSD versions predate the GNU versions and have fewer options. For each challenge program, I use a shell script to redirect the output from the original programs into output files. The goal is then to have the Rust program create the same output for the same inputs. I've been careful to include files encoded on Windows as well as simple ASCII text mixed with Unicode characters to force my programs to deal with various ideas of line endings and characters in the same way as the original programs.

For most of the challenges, I'll only try to implement a subset of the original programs as they can get pretty complicated. I also have chosen to make a few small changes in the output from some of the programs so that they are easier to teach. Consider this like learning to play an instrument by playing along with a recording. You don't have to play every note from the original version. The important thing is to learn common patterns like handling arguments and reading inputs so you can move on to writing your material.

Getting Rust and the Code

To start, you'll need to install Rust. One of my favorite parts about Rust is the `rustup` tool for installing, upgrading, and managing Rust. It works equally well on Windows and Unix-type operating systems (OS) like Linux and macOS. You will need to follow the the installation instructions for your OS. If you have already installed `rustup`, you might want to run **`rustup update`** to get the latest version of the language and tools as Rust updates about every six weeks. Execute **`rustup doc`** to read copious volumes of documentation. You can check the version of the `rustc` compiler with the following command:

```
$ rustc --version
rustc 1.54.0 (a178d0322 2021-07-26)
```

All the code, data, and tests for the programs can be found in my GitHub repository. You can use the Git source code management tool (which you may need to install) to get a copy of the code. The following command will create a new directory on your computer called *rust-sysprog* with the contents of the repository:

```
$ git clone https://github.com/kyclark/rust-sysprog.git
```

> **NOTE**
>
> You will *not* write your code in this directory. You should create a separate directory elsewhere for your projects, preferably in your own GitHub repository that you create for this purpose. You will copy each chapter's *tests* directory into your project directory to test your code.

One of the first tools you will encounter is Cargo, Rust's build tool, package manager, and test runner. You can use it to compile and test the code for Chapter 1. Change into the directory and run the tests with **`cargo test`**:

```
$ cd 01_hello
$ cargo test
```

If all goes well, you should see some passing tests (in no particular order):

```
running 3 tests
test false_not_ok ... ok
test true_ok ... ok
test runs ... ok
```

> **NOTE**
>
> I tested all the programs on macOS, Linux, Windows/Powershell, and Ubuntu Linux/Windows Subsystem for Linux (WSL). While I love how well Rust works on both Windows and Unix operating systems, two programs (`findr` and `lsr`) will not pass the entire test suite on Windows due to some fundamental differences in the operating system from Unix-type systems. I recommend Windows/Powershell users consider also installing WSL and working through the programs in that environment.

All the code in this book has been formatted using `rustfmt`, which is a really handy tool for making your code look pretty and readable. You can use **`cargo fmt`** to run it on all the source code in a project, or you can integrate it into your code editor to run on demand. For instance, I prefer to use the text editor `vim`, which I have configured to automatically run `rustfmt` every time I save my work. I find this makes it much easier to read my code and find mistakes.

I recommend you use Clippy, a linter for Rust code. *Linting* is automatically checking code for common mistakes, and it seems most languages offer one or more linters. Both `rustfmt` and `clippy` should be installed by default, but you can use **`rustup component add clippy`** if you need to install Clippy. Then you can run **`cargo clippy`** to have it check the source code and make recommendations. No output from Clippy means that it has no suggestions.

Now you're ready to write some Rust!

Conventions Used in This Book

The following typographical conventions are used in this book:

Italic

> Indicates new terms, URLs, email addresses, filenames, and file extensions.

`Constant width`

> Used for program listings, as well as within paragraphs to refer to program elements such as variable or function names, databases, data types, environment variables, statements, and keywords.

`Constant width bold`

> Shows commands or other text that should be typed literally by the user.

`Constant width italic`

Shows text that should be replaced with user-supplied values or by values determined by context.

> **TIP**
>
> This element signifies a tip or suggestion.

> **NOTE**
>
> This element signifies a general note.

> **WARNING**
>
> This element indicates a warning or caution.

Using Code Examples

Supplemental material (code examples, exercises, etc.) is available for download at *https://github.com/oreillymedia/title_title*.

If you have a technical question or a problem using the code examples, please send email to *bookquestions@oreilly.com*.

This book is here to help you get your job done. In general, if example code is offered with this book, you may use it in your programs and documentation. You do not need to contact us for permission unless you're reproducing a significant portion of the code. For example, writing a program that uses several chunks of code from this book does not require permission. Selling or distributing examples from O'Reilly books does require permission. Answering a question by citing this book and quoting example code does not require permission. Incorporating a significant amount of example code from this book into your product's documentation does require permission.

We appreciate, but generally do not require, attribution. An attribution usually includes the title, author, publisher, and ISBN. For example: “Systems Programming with Rust by Ken Youens-Clark (O’Reilly). Copyright 2021 Charles Kenneth Youens-Clark, 978-0-596-xxxx-x.”

If you feel your use of code examples falls outside fair use or the permission given above, feel free to contact us at *permissions@oreilly.com*.

O’Reilly Online Learning

> **NOTE**
>
> For more than 40 years, *O’Reilly Media* has provided technology and business training, knowledge, and insight to help companies succeed.

Our unique network of experts and innovators share their knowledge and expertise through books, articles, and our online learning platform. O’Reilly’s online learning platform gives you on-demand access to live training courses, in-depth learning paths, interactive coding environments, and a vast collection of text and video from O’Reilly and 200+ other publishers. For more information, visit *http://oreilly.com*.

How to Contact Us

Please address comments and questions concerning this book to the publisher:

O’Reilly Media, Inc.

1005 Gravenstein Highway North

Sebastopol, CA 95472

800-998-9938 (in the United States or Canada)

707-829-0515 (international or local)

707-829-0104 (fax)

We have a web page for this book, where we list errata, examples, and any additional information. You can access this page at *http://www.oreilly.com/catalog/9781098109424*.

Email *bookquestions@oreilly.com* to comment or ask technical questions about this book.

For news and information about our books and courses, visit *http://oreilly.com*.

Find us on Facebook: *http://facebook.com/oreilly*

Follow us on Twitter: *http://twitter.com/oreillymedia*

Watch us on YouTube: *http://www.youtube.com/oreillymedia*

Acknowledgments

I would like to thank my development editor, Corbin Collins, my production editor, <name here>. I am deeply indebted to the technical reviewers Carol Nichols, Brad Fulton, Erik Nordin, and Jeremy Gailor, as well as others who gave of their time to make comments including Joshua Lynch, Andrew Olson, and Jasper Zanjani.

Chapter 1. Truth Or Consequences

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 1st Chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the author at kyclark@gmail.com.

And the truth is we don't know anything

—They Might Be Giants

In this chapter, I'll show you how to organize, run, and test a Rust program. I'll be using a Unix platform (macOS) to explain some basic ideas about systems programs. Only some of these ideas apply to the Windows operating system, but the Rust programs themselves will work the same no matter which platform you use.

You will learn:

- How to compile Rust code into an executable
- How to use Cargo to start a new project
- About the `$PATH` environment variable

- How to use an external Rust crate from crates.io
- About the exit status of a program
- How to use common systems commands and options
- How write Rust versions of the `true` and `false` programs
- How to organize, write, and run tests

Getting Started with “Hello, world!”

The place to start is at the beginning, and it seems universally agreed that this means printing “Hello, world!” to the screen. You might change to a temporary directory with **`cd /tmp`** to write this first program, then fire up a text editor and type this Rust program into a file called *hello.rs*:

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

❶ Functions are defined using `fn`. The name of this function is `main`.

❷ `println!` (*print line*) is used to print to `STDOUT` (pronounced *standard out*). The semicolon indicates the end of the statement.

❸ The body of the function is enclosed in curly braces.

Rust will automatically start in the `main` function. Functions arguments appear inside the parentheses that follow. Because there are no arguments listed in `main()`, the function takes no arguments. The last thing I’ll point out here is `println!` looks like a function but is actually a macro, which is essentially code that writes code. You will see other macros in this book that also end with an exclamation point such as `assert!`.

To run this program, you must first use the Rust compiler `rustc` (or `rustc.exe` on Windows) to *compile* the code into a form that your computer can execute:

```
$ rustc hello.rs
```

If all goes well, there will be no output from the preceding command but you should now have a new file called *hello.exe* on Windows or *hello* on macOS and Linux. On macOS, I can use the `file` command to see what kind of file this is:

```
$ file hello
hello: Mach-O 64-bit executable x86_64
```

You should be able to execute this directly to see a charming and heartfelt message:

```
$ ./hello ❶
Hello, world!
```

❶ The dot (`.`) indicates the current directory.

> **TIP**
>
> Due to security issues, it's necessary to indicate an executable in the current directory using an explicit path.

On Windows, you can execute it like so:

```
> .\hello.exe
Hello, world!
```

That was cool.

Organizing a Rust Project Directory

In your Rust projects, you will likely write many files of source code and will also use code from sources like crates.io. It would be better to create a directory to contain all this. I'll create a directory called *hello*, and inside that

I'll create a *src* directory for the Rust *source* code files:

```
$ rm hello ❶
$ mkdir -p hello/src ❷
```

❶ Remove the `hello` binary so I can make a *hello* directory.

❷ The `mkdir` command will make a directory. The `-p` option says to create parent directories before creating child directories.

Now I'll move the *hello.rs* source file into *hello/src*:

```
$ mv hello.rs hello/src
```

Go into that directory and compile your program again:

```
$ cd hello
$ rustc src/hello.rs
```

You should again have a `hello` executable in the directory. I will use the `tree` command (which you might need to install) to show you the contents of my directory:

```
$ tree
.
├── hello
└── src
    └── hello.rs
```

This is the basic structure for a simple Rust project.

Creating and Running a Project with Cargo

It's actually easier to start a new Rust project if you use the Cargo tool that I introduced in the Preface. Move out of your *hello* directory and remove it:

```
$ cd .. ❶
$ rm -rf hello ❷
```

❶ The `cd` command will *change directories*. Two dots (`..`) indicate the parent directory.

❷ The `rm` command will remove a file or empty directory. The `-r` *recursive* option will remove the contents of a directory, and the `-f` *force* option will skip any errors.

Start your project anew using Cargo like so:

```
$ cargo new hello
     Created binary (application) `hello` package
```

This should create a new *hello* directory that you can change into. I'll use `tree` again to show you the contents:

```
$ cd hello
$ tree
.
├── Cargo.toml
└── src
    └── main.rs
```

> **NOTE**
>
> It's not a requirement to use Cargo to start a project, but it is highly encouraged and nearly everyone in the Rust community uses it. You can create the directory structure and files yourself if you prefer.

You can see that Cargo creates a *src* directory with the file *main.rs*. I'll use the command `cat` for *concatenate* (which you will write in Chapter 3) to show you the contents of this file, which should look familiar:

```
$ cat src/main.rs
fn main() {
    println!("Hello, world!");
}
```

Rather than using `rustc` to compile the program, I'd like you to use **`cargo`**

`run` to compile the source code and run it in one command:

```
$ cargo run
   Compiling hello v0.1.0 (/private/tmp/hello) ❶
    Finished dev [unoptimized + debuginfo] target(s) in 1.26s
     Running `target/debug/hello`
Hello, world! ❷
```

❶ The first three lines are information about what Cargo is doing.

❷ This is the output from the program.

If you would like for Cargo to be quieter, you can use the `-q` or `--quiet` option:

```
$ cargo run --quiet
Hello, world!
```

CARGO COMMANDS

How did I know about that option? If you run **`cargo`** with no other arguments, it will print the documentation which starts off like this:

```
$ cargo
Rust's package manager

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

OPTIONS:
    -V, --version                    Print version info and exit
        --list                       List installed commands
        --explain <CODE>             Run `rustc --explain CODE`
    -v, --verbose                    Use verbose output (-vv very
                                     verbose/build.rs output)
    -q, --quiet                      No output printed to stdout
        --color <WHEN>               Coloring: auto, always,
never
        --frozen                     Require Cargo.lock and cache
are up to date
        --locked                     Require Cargo.lock is up to
date
```

```
        --offline                    Run without accessing the
network
        --config <KEY=VALUE>...      Override a configuration
value (unstable)
    -Z <FLAG>...                     Unstable (nightly-only)
flags to Cargo,
                                     see cargo -Z help for
details
    -h, --help                       Prints help information
```

This gets at a key feature of good systems tools: They usually tell you how to use them, like how the cookie in *Alice in Wonderland* says “Eat me.” Notice that one of the first words in the preceding output is *USAGE*, so it’s common to call this the *usage*. The programs in this book will also print their usage. Read further to see that you can request *help* any of Cargo’s commands:

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

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

After running the program using Cargo, I can use the `ls` *list* command

(which you will write in Chapter 15) to see that there is now a new directory called *target*:

```
$ ls
Cargo.lock  Cargo.toml  src/        target/
```

You can use the `tree` command from earlier or the `find` command (which you will write in Chapter 7) to look at all the files that Cargo and Rust created. The executable file that ran should exist as *target/debug/hello*. You can run this directly just like the earlier program:

```
$ ./target/debug/hello
Hello, world!
```

So, without any guidance from us, Cargo managed to find the source code in *src/main.rs* and used the `main` function there to build a program called `hello` and then run it. Why was the binary file still called `hello`, though, and not `main`? To answer that, look at *Cargo.toml* [1], which is a configuration file for the project:

```
$ cat Cargo.toml
[package]
name = "hello" ❶
version = "0.1.0" ❷
edition = "2018" ❸

[dependencies]
```

❶ This was the name of the project I created with Cargo, so it will also be the name of the executable.

❷ This is the version of the program.

❸ This is the *edition* of Rust.

There are a couple of things to discuss here. First, Rust crates are expected to use *semantic version numbers* like `major.minor.patch` so that `1.2.4` is major version 1, minor version 2, patch version 4. Second, Rust editions

are a way the community introduces changes that are not backwards compatible. I will use the 2018 edition for all the programs in this book.

Writing and Running Integration Tests

> *More than the act of testing, the act of designing tests is one of the best bug preventers known. The thinking that must be done to create a useful test can discover and eliminate bugs before they are coded—indeed, test-design thinking can discover and eliminate bugs at every stage in the creation of software, from conception to specification, to design, coding, and the rest.*
>
> —Boris Beizer, Software Testing Techniques

A big part of what I hope to show you is the value of testing your code. As simple as this program is, there are still some things to verify. There are a couple of broad categories of tests I use which I might describe as *inside-out*, where I write tests for functions inside my program, and *outside-in*, where I write tests that run my programs as the user might. The first kind of tests are often called *unit* tests because functions are a basic unit of programming. I'll show you how to write those in Chapter 2. For this program, I want to start with the latter kind of testing which is often called *integration* testing because it checks that the program works as a whole.

It's common to create a *tests* directory for the integration test code. This keeps your source code nicely organized and also makes it easy for the compiler to ignore this code when you are not testing:

```
$ mkdir tests
```

I want to test the `hello` program by running it on the command line as the user will do, so I will create the file *tests/cli.rs* for *command line interface* (CLI). Your project should now look like this:

```
$ tree -L 2
.
├── Cargo.lock
```

```
├── Cargo.toml
├── src
│   └── main.rs
├── target
│   ├── CACHEDIR.TAG
│   └── debug
└── tests
    └── cli.rs
```

Start off by adding this function that shows the basic structure of a test in Rust:

```
#[test] ❶
fn works() {
    assert!(true); ❷
}
```

❶ The `#[test]` attribute tell Rust to run this function when testing.

❷ The `assert!` macro asserts that a Boolean expression is `true`.

All the tests in this book will use `assert!` to verify that some expectation is `true` or `assert_eq!` to verify that something is an expected value. Since this test is evaluating the literal value `true`, it will always succeed. To see this test in action, execute **`cargo test`**. You should see these lines among the output:

```
     Running tests/cli.rs (target/debug/deps/cli-27c6c9a94ed7c7df)

running 1 test
test works ... ok
```

To observe a failing test, change `true` to `false`:

```
#[test]
fn works() {
    assert!(false);
}
```

Among the output, you should see the following failed test:

```
running 1 test
test works ... FAILED
```

> **TIP**
>
> You can have as many `assert!` and `assert_eq!` calls in a test function as you like. If any of them fail, then the whole test fails.

Just asserting `true` and `false` is not useful, so remove that function. Instead I will see if the `hello` program can be executed. I will use `std::process::Command` to execute a command and check the result. To start, I'll demonstrate using the command `ls`, which I know works on both Unix and Windows:

```
use std::process::Command; ❶

#[test]
fn runs() {
    let mut cmd = Command::new("ls"); ❷
    let res = cmd.output(); ❸
    assert!(res.is_ok()); ❹
}
```

❶ Import the `std::process::Command` struct for creating a new command.

❷ Create a new command to run `ls`. Use `mut` to make this variable *mutable* as it will change.

❸ Run the command and capture the output which will be a `Result`.

❹ Verify that the result is an `Ok` value.

> **NOTE**
>
> By default, Rust variables are immutable, meaning their values cannot be changed.

Run **`cargo test`** and verify that you see a passing test among all the output:

```
running 1 test
test runs ... ok
```

Try changing the function to execute `hello` instead of `ls`:

```
#[test]
fn runs() {
    let mut cmd = Command::new("hello");
    let res = cmd.output();
    assert!(res.is_ok());
}
```

Run the tests again and note that it will fail because the `hello` program can't be found:

```
running 1 test
test runs ... FAILED
```

Recall that the binary exists in *target/debug/hello*. If you try to execute `hello` on the command line, you will see that the program can't be found:

```
$ hello
-bash: hello: command not found
```

When you execute any command, your operating system will look in a predefined set of directories for something by that name[2]. On Unix-type systems, you can inspect the `PATH` environment variable of your shell to see this list of directories that are delimited by colons. (On Windows, this is `$env:Path`.) I can use `tr` (*translate characters*) to replace the colons (`:`) with newlines (`\n`) to show you my `PATH`:

```
$ echo $PATH | tr : '\n' ❶
/opt/homebrew/bin
/Users/kyclark/.cargo/bin
/Users/kyclark/.local/bin
/usr/local/bin
```

```
/usr/bin
/bin
/usr/sbin
/sbin
```

❶ `$PATH` tells `bash` to interpolate the variable. Use a pipe `|` to feed this to `tr`.

Even if I change into the *target/debug* directory, `hello` can't be found due to the aforementioned security restrictions that exclude the current working directory from your `PATH`:

```
$ cd target/debug/
$ ls hello
hello*
$ hello
-bash: hello: command not found
```

I still have to explicitly reference the path:

```
$ ./hello
Hello, world!
```

Adding a Project Dependency

Currently, the `hello` program only exists in the *target/debug* directory. If I copy it to any of the directories in my `PATH` (note that I include `$HOME/.local/bin` directory for private programs), I can execute it and run the test successfully. I don't want to copy my program to test it; rather, I want to test the program that lives in the current crate. I can use the crate `assert_cmd` to find the program in my crate directory. I first need to add this as a development dependency to *Cargo.toml*. This tells Cargo that I only need this crate for testing and benchmarking:

```
[package]
name = "hello"
version = "0.1.0"
edition = "2018"
```

```
[dependencies]

[dev-dependencies]
assert_cmd = "1"
```

I can then use this crate to create a `Command` that looks in the Cargo binary directories. The following test does not verify that the program produces the correct output, only that it appears to succeed. Update your `runs` function with this definition:

```
use assert_cmd::Command; ❶

#[test]
fn runs() {
    let mut cmd = Command::cargo_bin("hello").unwrap(); ❷
    cmd.assert().success(); ❸
}
```

❶ Import `assert_cmd::Command`.

❷ Create a `Command` to run `hello` in the current crate. This returns a `Result` which I assume is safe to `Result::unwrap` because the binary is found.

❸ Execute the program and use `Assert::success` to "ensure the command succeeded."

> **NOTE**
>
> I'll have more to say about the `Result` type in following chapters. For now, just know that this is a way to model something that could succeed or fail for which there are two possible variants, `Ok` and `Err`, respectively.

Understanding Program Exit Values

What does it mean for a program to exit successfully? Systems programs should report a final exit status to the operating system. The Portable Operating System Interface (POSIX) standards dictate that the standard exit

code is `0` to indicate success (think *zero* errors) and any number from 1 to 255 otherwise. I can show you this using the `bash` shell and the `true` command. Here is the manual page from **`man true`** for the version that exists on macOS:

```
TRUE(1)                   BSD General Commands Manual
TRUE(1)

NAME
     true -- Return true value.

SYNOPSIS
     true

DESCRIPTION
     The true utility always returns with exit code zero.

SEE ALSO
     csh(1), sh(1), false(1)

STANDARDS
     The true utility conforms to IEEE Std 1003.2-1992
(``POSIX.2'').

BSD                               June 27, 1991
BSD
```

As the documentation shows, this program does nothing successfully. If I run **`true`**, I see nothing, but I can inspect the `bash` variable `$?` to see the exit status of the most recent program:

```
$ true
$ echo $?
0
```

The `false` command is a corollary in that it “always exits with a nonzero exit code”:

```
$ false
$ echo $?
1
```

All the programs you will write in this book will be expected to return the value `0` when they terminate normally and a nonzero value when there is an error. I can write my own versions of `true` and `false` to show you how to do this. Start by creating a *src/bin* directory, then create *src/bin/true.rs* with the following contents:

```
$ cat src/bin/true.rs
fn main() {
    std::process::exit(0); ❶
}
```

❶ Use the `std::process::exit` function to exit the program with the value `0`.

You're *src* directory should now have the following structure:

```
$ tree src/
src/
├── bin
│   └── true.rs
└── main.rs
```

You can run the program and manually check the exit value:

```
$ cargo run --quiet --bin true ❶
$ echo $?
0
```

❶ The `--bin` option is the "Name of the bin target to run."

Add the following test to *tests/cli.rs* to ensure it works correctly:

```
#[test]
fn true_ok() {
    let mut cmd = Command::cargo_bin("true").unwrap();
    cmd.assert().success();
}
```

If you run **`cargo test`**, you should see the following output:

```
running 2 tests
test true_ok ... ok
test runs ... ok
```

> **NOTE**
>
> The tests are not necessarily run in the same order they are declared in the code. This is because Rust is a safe language for writing *concurrent* code, which means code can be run across multiple threads. The testing takes advantage of this concurrency to run many tests in parallel, so the test results may appear in a different order each time you run them. This is a feature, not a bug. If you would like to run the tests in order, you can run them on a single thread via **`cargo test -- -- test-threads=1`**.

Rust programs will exit with the code `0` by default. Recall that *src/main.rs* doesn't explicitly call `std::process::exit`. This means that the `true` program can do nothing at all. Want to be sure? Change *src/bin/true.rs* to the following:

```
$ cat src/bin/true.rs
fn main() {}
```

Run the test suite and verify it still passes. Next, let's write a version of the `false` program with the following source code:

```
$ cat src/bin/false.rs
fn main() {
    std::process::exit(1); ❶
}
```

❶ Exit with any value from 1-255 to indicate an error.

Verify this looks OK to you:

```
$ cargo run --quiet --bin false
$ echo $?
1
```

Add this test to *tests/cli.rs* to verify that the program fails:

```
#[test]
fn false_not_ok() {
    let mut cmd = Command::cargo_bin("false").unwrap();
    cmd.assert().failure(); ❶
}
```

❶ Use the `Assert::failure` function to "ensure the command failed."

Run **`cargo test`** to verify that the programs all work as expected:

```
running 3 tests
test runs ... ok
test true_ok ... ok
test false_not_ok ... ok
```

Another way to write this program uses `std::process::abort`:

```
$ cat src/bin/false.rs
fn main() {
    std::process::abort();
}
```

Again, run the test suite to ensure that the program still works as expected.

Testing the Program Output

While it's nice to know that my `hello` program exits correctly, I'd like to ensure it actually prints the correct output to `STDOUT` (pronounced *standard out*), which is the standard place for output to appear and is usually the console. Update your `runs` function in *tests/cli.rs* to the following:

```
#[test]
fn runs() {
    let mut cmd = Command::cargo_bin("hello").unwrap();
    cmd.assert().success().stdout("Hello, world!\n"); ❶
}
```

❶ Verify that the command exits successfully and prints the given text to `STDOUT`.

Run the tests and verify that `hello` does, indeed, work correctly. Next, change *src/main.rs* to add some more exclamation points:

```
$ cat src/main.rs
fn main() {
    println!("Hello, world!!!");
}
```

Run the tests again to observe a failing test:

```
running 3 tests
test true_ok ... ok
test false_not_ok ... ok
test runs ... FAILED

failures:

---- runs stdout ----
thread runs panicked at 'Unexpected stdout, failed diff var
original
├── original: Hello, world!

├── diff:
--- value       expected
+ value actual
@@ -1 +1 @@
-Hello, world!
+Hello, world!!!

└── var as str: Hello, world!!!
```

Learning to read test output is a skill in itself, but the output is trying very hard to show you what was *expected* output and what was the *actual* output. While this is a trivial program, I hope you can see the value in automatically checking all aspects of the program we write.

Exit Values Make Programs Composable

Correctly reporting the exit status is a characteristic of well-behaved systems programs. The exit value is important because a failed process used in conjunction with another process should cause the combination to fail. For instance, I can use the logical *and* operator `&&` in `bash` to chain the two commands `true` and `ls`. Only if the first process reports success will the second process run:

```
$ true && ls
Cargo.lock  Cargo.toml  src/        target/     tests/
```

If instead I execute **`false && ls`**, the result is that the first process fails and `ls` is never executed. Additionally, the exit status of the whole command is nonzero.

```
$ false && ls
$ echo $?
1
```

Systems programs that correctly report errors make them composable with other programs. This is important because it's extremely common in Unix environments to combine many small commands to make *ad hoc* programs on the command line. If a program encounters an error but fails to report it to the operating system, then the results could be incorrect. It's far better for a program to abort so that the underlying problems can be fixed.

Summary

This chapter was meant to introduce you to some key ideas about organizing a Rust project and some basic ideas about systems programs. Here are some of the things you should understand:

- The Rust compiler `rustc` can compile Rust source code into a machine-executable file on Windows, macOS, and Linux.
- The Cargo tool can help create a Rust project. You can also use it to compile, run, and test the code.

- You saw several examples of systems tools like `ls`, `cd`, `mkdir`, and `rm` that accept arguments like file or directory name as well as options like `-f` or `-p`.
- POSIX-compatible programs should exit with a value of `0` to indicate success and any value 1-255 to indicate an error.
- You wrote the `hello` program to say "Hello, world!" and saw how to test it for the exit status and the text it printed to `STDOUT`.
- You learned to add crate dependencies to *Cargo.toml* and use the crates in your code.
- You created a *tests* directory to organize testing code, and you used `#[test]` to make functions that should be as tests.
- You learned how to write, run, and test alternate binaries in a Cargo project by creating souce code files in the *src/bin* directory.
- You wrote your own implementations of the `true` and `false` programs along with tests to verify that they succeed and fail as expected. You saw that by default a Rust program will exit with the value `0` and that the `std::process::exit` function can be used to explicitly exit with a given code. Additionally, the `std::process::abort` function can be used to exit with an error code.

---

1 TOML stands for Tom's Obvious, Minimal Language.

2 Shell aliases and functions can also be executed like commands, but I'm only talking about finding programs to run at this point.

Chapter 2. Test for Echo

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 2nd Chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the author at kyclark@gmail.com.

> *By the time you get this note, we'll no longer be alive, we'll have all gone up in smoke, there'll be no way to reply.*
>
> —They Might Be Giants

In Chapter 1, you wrote programs that always behave the same way. In this chapter, you will learn how to use arguments from the user to change the behavior of the program at runtime. The challenge program you'll write in this chapter is a clone of `echo`, which will print its arguments on the command line, optionally terminated with a newline.

In this chapter, you'll learn:

- How to process command-line arguments with the `clap` crate
- About `STDOUT` and `STDERR`
- About Rust types like strings, vectors, slices, and the unit type
- How to use expressions like `match`, `if`, and `return`

- How to use `Option` to represent an optional value
- How to handle errors using the `Result` variants of `Ok` and `Err`
- How to exit a program and signal success or failure
- The difference between stack and heap memory

Starting a New Binary Program with Cargo

This challenge program will be called `echor` for `echo` plus `r` for Rust. (I can't decide if I pronounce this like *eh-core* or *eh-koh-ar*.) I'm hoping you have created a directory to hold all the projects you'll write in this book. Change into that directory and use Cargo to start your *crate*, which is a Rust binary program or library:

```
$ cargo new echor
     Created binary (application) `echor` package
```

You should see a familiar directory structure:

```
$ tree echor/
echor/
├── Cargo.toml
└── src
    └── main.rs

1 directory, 2 files
```

Go into the new *echor* directory and use Cargo to run the program:

```
$ cd echor
$ cargo run
Hello, world! ❶
```

❶ The default program always prints "Hello, world!"

You've already seen this program in Chapter 1, but I'd like to point out a

couple more things about the code:

```
$ cat src/main.rs
fn main() {
    println!("Hello, world!");
}
```

As you saw in Chapter 1, Rust will start the program by executing the `main` function in *src/main.rs*. Any arguments to the function are contained in the parentheses after the function name, so the empty parentheses here indicate the function takes no arguments. All functions return a value, and the return type may be indicated with an arrow and the type such as `-> u32` to say the function returns an unsigned 32-bit integer. The lack of any return type for `main` implies that the function returns what Rust calls the *unit* type. Note that the `println!` macro will automatically append a newline to the output, which is a feature I'll need to control when the user requests no terminating newline.

> **NOTE**
>
> The unit type is like an empty value and is signified with a set of empty parentheses `()`. The documentation says this "is used when there is no other meaningful value that could be returned." It's not quite like a null pointer or undefined value in other languages, a concept first introduced by Tony Hoare (no relational to Rust creator Graydon Hoare) who called the null reference his "billion-dollar mistake." Since Rust does not (normally) allow you to dereference a null pointer, it must logically be worth at least a billion dollars.

How echo Works

The `echo` program is blissfully simple. I'll review how it works so you can consider all the features your version will need. To start, `echo` will print its arguments to `STDOUT`:

```
$ echo Hello
Hello
```

I'm using the `bash` shell which assumes that any number of spaces delimit the arguments, so arguments that have spaces must be enclosed in quotes. In the following command, I'm providing four words as a single argument:

```
$ echo "Rust has assumed control"
Rust has assumed control
```

Without the quotes, I'm providing four separate arguments. Note that I use a varying number of spaces when I provide the arguments, but `echo` prints them using a single space between each argument:

```
$ echo Rust  has assumed   control
Rust has assumed control
```

If I want the spaces to be preserved, I must enclose them in quotes:

```
$ echo "Rust  has assumed   control"
Rust  has assumed   control
```

It's extremely common for command-line programs to respond to the flags `-h` or `--help` to print a message about how to use the program, the so-called *usage* because that is usually the first word of the output. If I try that with `echo`, I just get back the text of the flag:

```
$ echo --help
--help
```

To understand more about the program, execute **`man echo`** to read the manual page. You'll see that I'm using the BSD version of the program from 2003:

```
ECHO(1)                   BSD General Commands Manual
ECHO(1)

NAME
     echo -- write arguments to the standard output

SYNOPSIS
```

```
     echo [-n] [string ...]

DESCRIPTION
     The echo utility writes any specified operands, separated by
single blank
     (' ') characters and followed by a newline ('\n') character,
to the stan-
     dard output.

     The following option is available:

     -n    Do not print the trailing newline character.  This may
also be
           achieved by appending '\c' to the end of the string, as
is done by
           iBCS2 compatible systems.  Note that this option as well
as the
           effect of '\c' are implementation-defined in IEEE Std
1003.1-2001
           (''POSIX.1'') as amended by Cor. 1-2002.  Applications
aiming for
           maximum portability are strongly encouraged to use
printf(1) to
           suppress the newline character.

     Some shells may provide a builtin echo command which is
similar or iden-
     tical to this utility.  Most notably, the builtin echo in
sh(1) does not
     accept the -n option.  Consult the builtin(1) manual page.

EXIT STATUS
     The echo utility exits 0 on success, and >0 if an error
occurs.

SEE ALSO
     builtin(1), csh(1), printf(1), sh(1)

STANDARDS
     The echo utility conforms to IEEE Std 1003.1-2001
(''POSIX.1'') as
     amended by Cor. 1-2002.

BSD                              April 12, 2003
BSD
```

By default, the text printed on the command line is terminated by a newline

character. As shown in the preceding manual page, `echo` has a single option `-n` that will omit the newline. Depending on the version of `echo` you have, this may not appear to affect the output. For instance, the BSD version I'm using shows this:

```
$ echo -n Hello
Hello
$ ❶
```

❶ The BSD `echo` shows my command prompt `$` on the next line.

While the GNU version on Linux shows this:

```
$ echo -n Hello
Hello$ ❶
```

❶ The GNU `echo` shows my command prompt immediately after *Hello*.

Regardless which version of `echo`, I can use the `bash` redirect operator `>` to send `STDOUT` to a file:

```
$ echo Hello > hello
$ echo -n Hello > hello-n
```

The `diff` tool will display the *differences* between two files. This output shows that the second file (*hello-n*) does not have a newline at the end:

```
$ diff hello hello-n
1c1
< Hello
---
> Hello
\ No newline at end of file
```

Getting Command-Line Arguments

The first order of business is getting the command-line arguments to print. In

Rust you can use `std::env::args` for this. The `std` tells me this is in the *standard* library, and is Rust code that is so universally useful it included with the language. The `env` part tells me this is for interacting with the *environment*, which is where the program will find the arguments. If you look at the documentation for the function, you'll see it returns something of the type `Args`:

```
pub fn args() -> Args
```

If you follow the link for the `Args` documentation, you'll find it is a *struct*, which is a kind of data structure in Rust. If you look along the left-hand side of the page, you'll see things like trait implementations, other related structs, functions, and more. I'll explore these ideas later, but for now, just poke around the docs and try to absorb what you see.

Edit *src/main.rs* to print the arguments. I can call the function by using the full path followed by an empty set of parentheses:

```
fn main() {
    println!(std::env::args()); // This will not work
}
```

When I execute the program using **`cargo run`**, I see the following error:

```
$ cargo run
   Compiling echor v0.1.0 (/Users/kyclark/work/sysprog-
rust/playground/echor)
error: format argument must be a string literal
 --> src/main.rs:2:14
  |
2 |     println!(std::env::args()); // This will not work
  |              ^^^^^^^^^^^^^^^^
  |
help: you might be missing a string literal to format with
  |
2 |     println!("{}", std::env::args()); // This will not work
  |              ^^^^^
```

Here is my first spat with the compiler. It's saying that it cannot directly print

the value that is returned from that function, but it's also telling me exactly how to fix the problem. It wants me to first provide a literal string that has a set of curly brackets `{}` that will serve as a placeholder for the printed value, so I change my code accordingly:

```
fn main() {
    println!("{}", std::env::args()); // This will not work either
}
```

Run the program again and see that we're not out of the woods yet. Note that I will omit the "Compiling" and other lines to focus on the important output:

```
$ cargo run
error[E0277]: `Args` doesn't implement `std::fmt::Display`
 --> src/main.rs:2:20
  |
2 |     println!("{}", std::env::args()); // This will not work
either
  |                    ^^^^^^^^^^^^^^^^ `Args` cannot be formatted
with
                                        the default formatter
  |
  = help: the trait `std::fmt::Display` is not implemented for
`Args`
  = note: in format strings you may be able to use `{:?}` (or {:#?}
for
    pretty-print) instead
  = note: required by `std::fmt::Display::fmt`
  = note: this error originates in a macro (in Nightly builds, run
with
    -Z macro-backtrace for more info)
```

There's a lot of information in that compiler message. First off, there's something about the trait `std::fmt::Display` not being implemented for `Args`. A *trait* in Rust is a way to define the behavior of an object in an abstract way. If an object implements the `Display` trait, then it can be formatted for "user-facing output." Look again at the "Trait Implementations" section of the `Args` documentation and notice that, indeed, `Display` is not mentioned there.

The compiler suggests I should use `{:?}` instead of `{}` for the placeholder.

This is an instruction to print a `Debug` version of the structure, which will "format the output in a programmer-facing, debugging context." Refer again to the `Args` documentation to see that `Debug` is listed under "Trait Implementations," so works:

```
fn main() {
    println!("{:?}", std::env::args()); // Success at last!
}
```

Now the program compiles and prints something vaguely useful:

```
$ cargo run
Args { inner: ["target/debug/echor"] }
```

If you are unfamiliar with command-line arguments, it's common for the first value to be the path of the program itself. It's not an argument *per se*, but it is useful information. One thing to note is that this program was compiled into the path *target/debug/echor*. Unless you state otherwise, this is where Cargo will place the executable, also called a *binary*. Next, I'll pass some arguments:

```
$ cargo run Hello world
Args { inner: ["target/debug/echor", "Hello", "world"] }
```

Huzzah! It would appear that I'm able to get the arguments to my program. I passed two arguments, *Hello* and *world*, and they showed up as additional values after the binary name. I know I'll need to pass the `-n` flag, so let me try that:

```
$ cargo run Hello world -n
Args { inner: ["target/debug/echor", "Hello", "world", "-n"] }
```

It's also common to place the flag before the values, so let me try that:

```
$ cargo run -n Hello world
error: Found argument '-n' which wasn't expected, or isn't valid in
this context
```

```
USAGE:
    cargo run [OPTIONS] [--] [args]...

For more information try --help
```

That doesn't work because Cargo thinks the `-n` argument is for itself, not the program I'm running. To fix this, I need to separate Cargo's options using two dashes:

```
$ cargo run -- -n Hello world
Args { inner: ["target/debug/echor", "-n", "Hello", "world"] }
```

In the parlance of program parameters, the `-n` is an *optional* argument because you can leave it out. Typically program options start with one or two dashes. It's common to have *short* names with one dash and a single character like `-h` for the *help* flag and *long* names with two dashes and a word like `--help`. Specifically, `-n` and `-h` are *flags* that have one meaning when present and the opposite when absent. In this case, `-n` says to omit the trailing newline; otherwise, print as normal.

All the other arguments to `echo` are *positional* because their position relative to the name of the program (the first element in the arguments) determines their meaning. Consider the command `chmod` that takes two positional arguments, a mode like `755` first and a file or directory name second. In the case of `echo`, all the positional arguments are interpreted as the text to print, and they should be printed in the same order they are given. This is not a bad start, but the arguments to the programs in this book are going to become much more complex. I need to find a better way to parse the program's arguments.

Adding clap as a Dependency

Although there are various methods and crates for parsing command-line arguments, I will exclusively use the `clap` crate in this book. To get started, I need to tell Cargo that I want to download this crate and use it in my project. I can do this by adding a dependency to *Cargo.toml*. Edit your file to

add `clap` version 2.33 to the `[dependencies]` section:

```
$ cat Cargo.toml
[package]
name = "echor"
version = "0.1.0"
edition = "2018"

[dependencies]
clap = "2.33" ❶
```

❶ The crate name is not quoted. The equal sign indicates the version of the crate, which should be quoted.

> **NOTE**
>
> The version "2.33" means I want to use exactly this version. I could use just "2" to indicate that I'm fine using the latest version in the "2.x" line. There are many other ways to indicate the version, and I recommend you read how to specify dependencies.

The next time I try to build the program, Cargo will download the `clap` source code (if needed) and all of its dependencies. For instance, I can run **`cargo build`** to just build the new binary and not run it:

```
$ cargo build
    Updating crates.io index
   Compiling libc v0.2.93
   Compiling bitflags v1.2.1
   Compiling unicode-width v0.1.8
   Compiling vec_map v0.8.2
   Compiling strsim v0.8.0
   Compiling ansi_term v0.11.0
   Compiling textwrap v0.11.0
   Compiling atty v0.2.14
   Compiling clap v2.33.3
   Compiling echor v0.1.0 (/Users/kyclark/work/sysprog-
rust/playground/echor)
    Finished dev [unoptimized + debuginfo] target(s) in 27.30s
```

You may be curious where these packages went. Cargo places the download

source code into `$HOME/.cargo`, and the build artifacts go into the *target/debug/deps* directory. This brings up an interesting part of building Rust projects: Each program you build can use different versions of crates, and each program is built in a separate directory. If you have ever suffered through using shared modules as is common with Perl and Python, you'll appreciate that you don't have to worry about conflicts where one program requires some old obscure version and another requires the latest bleeding-edge version in GitHub. Python, of course, offers *virtual environments* to combat this problem, and other languages have similar solutions. Still, I find Rust's approach to be quite comforting.

A consequence of Rust placing the dependencies into the *target* directory is that it's now quite large. I'm already at close to 30MB for the project directory, with almost all of that living in the *target* directory. If you run **`cargo help`**, you will see that the `clean` command will remove the *target* directory at the expense of having to recompile again in the future. You might do to reclaim disk space if you aren't going to work on the project for a while.

Parsing Command-Line Arguments Using clap

To learn how to use `clap` to parse the arguments, I need to read the documentation. I like to use Docs.rs, "an open source documentation host for crates of the Rust Programming Language." I can follow examples from the documentation that show how to create a new `App` struct. Change your *src/main.rs* to the following:

```
use clap::App; ❶

fn main() {
    let _matches = App::new("echor") ❷
        .version("0.1.0") ❸
        .author("Ken Youens-Clark <kyclark@gmail.com>") ❹
        .about("Rust echo") ❺
        .get_matches(); ❻
}
```

❶ Import the `clap::App` struct.

❷ Create a new `App` with the name *echor*.

❸ Use semantic version numbers as described earlier.

❹ Your name and email address so people know where to send the money.

❺ This is a short description of the program.

❻ Ask the `App` to parse the arguments.

> **NOTE**
>
> In the preceding code, the leading underscore in the variable name `_matches` is functional. It tells the Rust compiler that I do not intend to use this variable right now. Without the underscore, the compiler would warn about an unused variable.

With this code in place, I can run the program with the `-h` or `--help` flags to get a usage document. Note that I didn't have to define this argument as `clap` did this for me:

```
$ cargo run -- -h
echor 0.1.0 ❶
Ken Youens-Clark <kyclark@gmail.com> ❷
Rust echo ❸

USAGE:
    echor

FLAGS:
    -h, --help       Prints help information
    -V, --version    Prints version information
```

❶ The app name and `version` number appear here.

❷ Here is the `author` information.

❸ This is the `about` text.

In addition to the help flags, I see that `clap` also automatically handles the

flags `-V` and `--version` to print the program's version:

```
$ cargo run -- --version
echor 0.1.0
```

Next, I need to define the parameters which I can do by adding `Arg` structs to the `App`.

```
use clap::{App, Arg}; ❶

fn main() {
    let matches = App::new("echor")
        .version("0.1.0")
        .author("Ken Youens-Clark <kyclark@gmail.com>")
        .about("Rust echo")
        .arg( ❷
            Arg::with_name("text")
                .value_name("TEXT")
                .help("Input text")
                .required(true)
                .min_values(1),
        )
        .arg( ❸
            Arg::with_name("omit_newline")
                .help("Do not print newline")
                .takes_value(false)
                .short("n"),
        )
        .get_matches(); ❹

    println!("{:#?}", matches); ❺
}
```

❶ Import both the `App` and `Arg` structs from the `clap` crate.

❷ Create a new `Arg` with the name `text`. This is a required positional argument that must appear at least once and can be repeated.

❸ Create a new `Arg` with the name `omit_newline`. This is a flag that has only the short name `-n` and takes no value.

❹ Parse the arguments and return the matching elements.

❺ Pretty-print the arguments.

> **NOTE**
>
> Earlier I used `{:?}` to format the debug view of the arguments. Here I'm using `{:#?}` to include newlines and indentations to help me read the output. This is called *pretty-printing* because, well, it's prettier.

If you request the usage again, you will see the new parameters:

```
$ cargo run -- --help
echor 0.1.0
Ken Youens-Clark <kyclark@gmail.com>
Rust echo

USAGE:
    echor [FLAGS] <TEXT>...

FLAGS:
    -h, --help       Prints help information
    -n               Do not print newline ❶
    -V, --version    Prints version information

ARGS:
    <TEXT>...    Input text ❷
```

❶ The `-n` flag to omit the newline is optional.

❷ The required input text is one or more positional arguments.

Run the program with some arguments and inspect the structure of the arguments:

```
$ cargo run -- -n Hello world
ArgMatches {
    args: {
        "text": MatchedArg {
            occurs: 2,
            indices: [
                2,
                3,
            ],
            vals: [
                "Hello",
```

```
                "world",
            ],
        },
        "omit_newline": MatchedArg {
            occurs: 1,
            indices: [
                1,
            ],
            vals: [],
        },
    },
    subcommand: None,
    usage: Some(
        "USAGE:\n    echor [FLAGS] <TEXT>...",
    ),
}
```

If you run the program with no arguments, you will get an error indicating that you failed to provide the required arguments:

```
$ cargo run
error: The following required arguments were not provided:
    <TEXT>...

USAGE:
    echor [FLAGS] <TEXT>...

For more information try --help
```

This was an error, and so you can inspect the exit value to verify that it’s not 0:

```
$ echo $?
1
```

If you try to provide any argument that isn’t defined, it will trigger an error and a nonzero exit value:

```
$ cargo run -- -x
error: Found argument '-x' which wasn't expected, or isn't valid in
this context

USAGE:
```

```
    echor [FLAGS] <TEXT>...

For more information try --help
```

> **NOTE**
>
> You might wonder how this magical stuff is happening. Why is the program stopping and reporting these errors? If you read the documentation for `App::get_matches`, you'll see that "upon a failed parse an error will be displayed to the user and the process will exit with the appropriate error code."

There's another subtle thing happening with the output that is not at all obvious. The usage and error messages are all appearing on STDERR (pronounced *standard error*), which is another channel of Unix output. To see this in the `bash` shell, I can redirect channel `1` (STDOUT) to a file called *out* and channel `2` (STDERR) to a file called *err*:

```
$ cargo run 1>out 2>err
```

You should see no results from that command because all the output was redirected to files. The *out* file should be empty because there was nothing printed to STDOUT, but the *err* file should contain the output from Cargo and the error messages from the program:

```
$ cat err
    Finished dev [unoptimized + debuginfo] target(s) in 0.01s
     Running `target/debug/echor`
error: The following required arguments were not provided:
    <TEXT>...

USAGE:
    echor [FLAGS] <TEXT>...

For more information try --help
```

So you see that another hallmark of well-behaved systems programs is to print regular output to STDOUT and error messages to STDERR. Sometimes errors are severe enough that you should halt the program, but sometimes

they should just be noted in the course of running. For instance, in Chapter 3 you will write a program that processes input files, some of which will intentionally not exist or will be unreadable. I will show you how to print warnings to STDERR about these files and skip to the next argument.

Creating the Program Output

My next step is to use the values provided by the user to create the program's output. It's common to copy the values out of the `matches` into variables. To start, I want to extract the `text` argument. Because this `Arg` was defined to accept one or more values, I can use either of these functions that return multiple values:

- `ArgMatches::values_of`: returns `Option<Values>`
- `ArgMatches::values_of_lossy`: returns `Option<Vec<String>>`

To decide which to use, I have to run down a few rabbit holes to understand the following concepts:

- `Option`: a value that is either `None` or `Some(T)` where `T` is any *type* like a string or an integer. In the case of `ArgMatches::values_of_lossy`, the type `T` will be a vector of strings.
- `Values`: An iterator for getting multiple values out of an argument.
- `Vec`: A vector, which is a contiguous growable array type.
- `String`: A string of characters.

Both of the functions `ArgMatches::values_of` and `ArgMatches::values_of_lossy` will return an `Option` of something. Since I ultimately want to print the strings, I will use `ArgMatches::values_of_lossy` function to get an `Option<Vec<String>>`. The `Option::unwrap` function will take the

value out of `Some(T)` to get at the payload `T`. Because the `text` argument is required by `clap`, I know it will be impossible to have `None`; therefore, I can safely call `Option::unwrap` to get the `Vec<String>` value:

```
let text = matches.values_of_lossy("text").unwrap();
```

> **WARNING**
>
> If you call `Option::unwrap` on a `None`, it will cause a *panic* that will crash your program. You should only call `unwrap` if you are positive the value is the `Some` variant.

The `omit_newline` argument is a bit easier as it's either present or not. The type of this value will be a `bool` or Boolean, which is either `true` or `false`:

```
let omit_newline = matches.is_present("omit_newline");
```

Finally, I want to print the values. Because `text` is a vector of strings, I can use `Vec::join` to join all the strings on a single space into a new string to print. Inside the `echor` program, `clap` will be creating the vector. To demonstrate how `Vec::join` works, I'll show you how create a vector using the `vec!` macro:

```
let text = vec!["Hello", "world"];
```

> **NOTE**
>
> The values in Rust vectors must all be of the same type. Dynamic languages often allow lists to mix types like strings and numbers, but Rust will complain about "mismatched types." Here I want a list of literal strings which must be enclosed in double quotes. The `str` type in Rust represents a valid UTF-8 string. I'll have more to say about UTF in Chapter 4.

`Vec::join` will insert the given string between all the elements of the vector to create a new string. I can use `println!` to print the new string to

STDOUT followed by a newline:

```
println!("{}", text.join(" "));
```

It's common practice in Rust documentation to present facts using `assert!` to say that something is `true` or `assert_eq!` to demonstrate that one thing is equivalent to another. In the following code, I can assert that the result of `text.join(" ")` is equal to the string `"Hello world"`:

```
assert_eq!(text.join(" "), "Hello world");
```

When the `-n` flag is present, the output should omit the newline. I will instead use the `print!` macro which does not add a newline, and I will choose to add either a newline or the empty string depending on the value of `omit_newline`. Depending on your background, you might try to write something like this:

```
let ending = "\n"; ❶
if omit_newline {
    ending = ""; // This will not work ❷
}
print!("{}{}", text.join(" "), ending); ❸
```

❶ Assign a default value.

❷ Change the value if the newline should be omitted.

❸ Use `print!` which will not add a newline to the output.

If I try to run this code, Rust tells me that I cannot reassign the value of `ending`:

```
$ cargo run -- Hello world
error[E0384]: cannot assign twice to immutable variable `ending`
  --> src/main.rs:27:9
   |
25 |     let ending = "\n";
   |         ------
   |         |
```

```
   |         first assignment to `ending`
   |         help: make this binding mutable: `mut ending`
26 |     if omit_newline {
27 |         ending = ""; // This will not work
   |         ^^^^^^^^^^^ cannot assign twice to immutable variable
```

Something that really sets Rust apart from other languages is that variables are *immutable* by default, meaning they can't be altered from their initial value. I'm not allowed to reassign the value of the variable `ending`; however, the compiler tells me to add `mut` to make the variable *mutable*:

```
let mut ending = "\n"; ❶
if omit_newline {
    ending = "";
}
print!("{}{}", text.join(" "), ending);
```

❶ Add `mut` to make this a mutable value.

There's a much better way to write this. In Rust, `if` is an expression and not a statement. An *expression* returns a value, but a statement does not. Here's a more Rustic way to write this:

```
let ending = if omit_newline { "" } else { "\n" };
```

> **NOTE**
>
> An `if` without an `else` will return the unit type. The same is true for a function without a return type, so the `main` function returns `()`.

Since I only use `ending` in one place, I don't need to assign it to a variable. Here is how I would update the `main` function:

```
fn main() {
    let matches = ...; // Same as before
    let text = matches.values_of_lossy("text").unwrap();
    let omit_newline = matches.is_present("omit_newline");
    print!("{}{}", text.join(" "), if omit_newline { "" } else {
```

```
"\n" });
}
```

With these changes, the program appears to work correctly; however, I'm not willing to stake my reputation on this. I need to, as the Russian saying goes, "Доверяй, но проверяй."[1] This requires that I write some tests to run my program with various inputs and verify that it produces the same output as the original `echo` program.

Integration and Unit Tests

In Chapter 1, I showed how to create *integration* tests that run the program from the command line just as the user will do to ensure it works correctly. In this chapter, I'll also show you how to write *unit* tests that exercise individual functions, which might be considered a unit of programming. To get started, you should add the following dependencies to *Cargo.toml*. Note that I'm adding `predicates` to this project:

```
[dev-dependencies]
assert_cmd = "1"
predicates = "1"
```

I often write tests that ensure my programs fail when run incorrectly. For instance, this program ought to fail and print help documentation when provided no arguments. Create a *tests* directory, and then create *tests/cli.rs* with the following:

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

#[test]
fn dies_no_args() {
    let mut cmd = Command::cargo_bin("echor").unwrap();
    cmd.assert() ❷
        .failure()
        .stderr(predicate::str::contains("USAGE"));
}
```

❶ Import the `predicates` crate.

❷ Run the program with no arguments and assert that it fails and prints a usage statement to `STDERR`.

> **NOTE**
>
> I usually name these sorts of tests with the prefix *dies* so that I can run them all with **`cargo test dies`** to ensure the program fails under various conditions.

I can also add a test to ensure the program exits successfully when provided an argument:

```
#[test]
fn runs() {
    let mut cmd = Command::cargo_bin("echor").unwrap();
    cmd.arg("hello").assert().success(); ❶
}
```

❶ Run `echor` with the argument "hello" and verify it exits successfully.

Creating the Test Output Files

I can now run **`cargo test`** to verify that I have a program that runs, validates user input, and prints usage. Next, I would like to ensure that the program creates the same output as `echo`. To start, I need to capture the output from the original `echo` for various inputs so that I can compare these to the output from my program. In the *02_echor* directory of my GitHub repository, you'll find a `bash` script called *mk-outs.sh* that I used to generate the output from `echo` for various arguments. You can see that, even with such a simple tool, there's still a decent amount of *cyclomatic complexity*, which refers to the various ways all the parameters can be combined. I need to check one or more text arguments both with and without the newline option:

```
$ cat mk-outs.sh
```

```
#!/usr/bin/env bash ❶

OUTDIR="tests/expected" ❷
[[ ! -d "$OUTDIR" ]] && mkdir -p "$OUTDIR" ❸

echo "Hello there" > $OUTDIR/hello1.txt ❹
echo "Hello"  "there" > $OUTDIR/hello2.txt ❺
echo -n "Hello  there" > $OUTDIR/hello1.n.txt ❻
echo -n "Hello" "there" > $OUTDIR/hello2.n.txt ❼
```

❶ The “shebang” line tells the operating system to use the environment to execute `bash` for the following code.

❷ Define a variable for the output directory.

❸ Test if the output directory does not exist and create it if needed.

❹ One argument with two words.

❺ Two arguments separated by more than one space.

❻ One argument with two spaces and no newline.

❼ Two arguments with no newline.

If you are working on a Unix platform, you can copy this program to your project directory and run it like so:

```
$ bash mk-outs.sh
```

It’s also possible to execute the program directly, but you may need to execute **`chmod +x mk-outs.sh`** if you get a *permission denied* error:

```
$ ./mk-outs.sh
```

If this worked, you should now have a *tests/expected* directory with the following contents:

```
$ tree tests
tests
├── cli.rs
└── expected
    ├── hello1.n.txt
```

```
    ├── hello1.txt
    ├── hello2.n.txt
    └── hello2.txt

1 directory, 5 files
```

If you are working on a Windows platform, then I recommend you copy this directory structure into your project. Now you should have some test files to use in comparing the output from your program.

Comparing Program Output

The first output file was generated with the input *Hello there* as a single string, and the output was captured into the file *tests/expected/hello1.txt*. For my next test, I will run `echor` with this argument and compare the output to the contents of that file. Be sure add `use std::fs` to *tests/cli.rs* bring in the standard *file system* module, and then replace the `runs` function with this:

```
#[test]
fn hello1() {
    let outfile = "tests/expected/hello1.txt"; ❶
    let expected = fs::read_to_string(outfile).unwrap(); ❷
    let mut cmd = Command::cargo_bin("echor").unwrap(); ❸
    cmd.arg("Hello there").assert().success().stdout(expected); ❹
}
```

❶ This is the output from `echo` generated by `mk-outs.sh`.

❷ Use `fs::read_to_string` to read the contents of the file. This returns a `Result` that might contain a string if all goes well. Use the `Result::unwrap` method with the assumption that this will work.

❸ Create a `Command` to run `echor` in the current crate.

❹ Run the program with the given argument and assert it finishes successfully and that `STDOUT` is the expected value.

WARNING

The `fs::read_to_string` is a convenient way to read a file into memory, but it's also an easy way to crash your program—and possibly your computer—if you happen to read a file that exceeds your available memory. You should only use this function with small files. As Ted Nelson says, "The good news about computers is that they do what you tell them to do. The bad news is that they do what you tell them to do."

If you run **`cargo test`**, you might see output like this:

```
running 2 tests
test hello1 ... ok
test dies_no_args ... ok
```

Using the Result Type

I've been using the `Result::unwrap` method in such a way that assumes each fallible call will succeed. For example, in the `hello1` function, I assumed that the output file exists and can be opened and read into a string. During my limited testing, this may be the case, but it's dangerous to make such assumptions. I'd rather be more cautious, so I'm going to create a type alias called `TestResult`. This will be a specific type of `Result` that is either an `Ok` which always contains the unit type or some value that implements the `std::error::Error` trait:

```
type TestResult = Result<(), Box<dyn std::error::Error>>;
```

In the preceding code, `Box` indicates that the error will live inside a kind of pointer where the memory is dynamically allocated on the heap rather than the stack, and `dyn` "is used to highlight that calls to methods on the associated Trait are dynamically dispatched." That's really a lot of information, and I don't blame you if your eyes glazed over. In short, I'm saying that the `Ok` part of `TestResult` will only ever hold the unit type, and the `Err` part can hold anything that implements the `std::error::Error` trait. These concepts are more thoroughly explained in *Programming Rust* (O'Reilly, 2021).

STACK AND HEAP MEMORY

Before programming in Rust, I only ever considered one, amorphous idea of computer memory. Having studiously avoided languages that required me to allocate and free memory, I was only vaguely aware of the efforts that dynamic languages made to hide these complexities from me. In Rust, I've learned that not all memory is accessed in the same way. First, there is the *stack* where items of known sizes are accessed in a particular order. The classic analogy is to a stack of cafeteria trays where clean items go on top and are taken back off the top in *last in, first out* (LIFO) order. Items on the stack have a fixed, known size, making it possible for Rust to set aside a particular chunk of memory and find it quickly.

The other type of memory is *heap* where the size of the values may change over time. For instance, the documentation for vector describes this structure as a "contiguous growable array type." The *growable* is the key word as the number and size of the elements in a vector can change during the lifetime of the program. Rust make an estimation of the amount of memory it needs for the vector. If the vector grows beyond the original allocation, Rust will find another chunk of memory to hold the data. To find the memory where the data lives, Rust stores the memory address on the stack. This is called a *pointer* because it points to the actual data and so is also said to be a *reference* to the data. Rust knows how to *dereference* a `Box` to find the data.

This changes my test code in some subtle ways. All the functions now indicate that they return a `TestResult`. Previously I used `Result::unwrap` to unpack `Ok` values and panic in the event of an `Err`, causing the test to fail. In the following code, I replace `unwrap` with the `?` operator to either unpack an `Ok` value or propagate the `Err` value to the return type. That is, this will cause the function to return the `Err` variant of `Option` to the caller, which will in turn cause the test to fail. If all the code in a test function runs successfully, I return an `Ok` containing the unit type to indicate the test passes. Note that while Rust does have `return` to return a

value early from a function, the idiom is to omit the semicolon from the last expression to implicitly return that result. Update your *tests/cli.rs* to this:

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

type TestResult = Result<(), Box<dyn std::error::Error>>;

#[test]
fn dies_no_args() -> TestResult {
    let mut cmd = Command::cargo_bin("echor")?; ❶
    cmd.assert()
        .failure()
        .stderr(predicate::str::contains("USAGE"));
    Ok(()) ❷
}

#[test]
fn hello1() -> TestResult {
    let expected =
fs::read_to_string("tests/expected/hello1.txt")?;
    let mut cmd = Command::cargo_bin("echor")?;
    cmd.arg("Hello there").assert().success().stdout(expected);
    Ok(())
}
```

❶ Use `?` instead of `Result::unwrap` to unpack an `Ok` value or propagate an `Err`.

❷ Omit the final semicolon to return this value.

The next test passes two arguments, *Hello* and *there*, and expects the program to print *Hello there*.

```
#[test]
fn hello2() -> TestResult {
    let expected =
fs::read_to_string("tests/expected/hello2.txt")?;
    let mut cmd = Command::cargo_bin("echor")?;
    cmd.args(vec!["Hello", "there"]) ❶
        .assert()
        .success()
        .stdout(expected);
```

```
    Ok(())
}
```

❶ Use the `Command::args` method to pass a vector of arguments rather than a single string value.

I have a total of four files to check, so it behooves me to write a helper function. I'll call it `run` and will pass it the argument strings along with the expected output file. Rather than use a vector for the arguments, I'm going to use a `std::slice` because I don't need to grow the argument list after I've defined it:

```
fn run(args: &[&str], expected_file: &str) -> TestResult { ❶
    let expected = fs::read_to_string(expected_file)?; ❷
    Command::cargo_bin("echor")? ❸
        .args(args)
        .assert()
        .success()
        .stdout(expected);
    Ok(()) ❹
}
```

❶ The `args` will be a slice of `&str` values, and the `expected_file` will be a `&str`. The return value is a `TestResult`.

❷ Try to read the contents of the `expected_file` into a string.

❸ Attempt to run `echor` in the current crate with the given arguments and assert that `STDOUT` is the expected value.

❹ If all the previous code worked, return `Ok` containing the unit type.

> **NOTE**
>
> You will find that Rust has many types of "string" variables. The type `str` is appropriate here for literal strings in the source code. The & shows that I intend only to borrow the string for a little while. I'll have more to say about strings, borrowing, and ownership later.

Below is how I can use the helper function to run all four tests. Replace the

earlier `hello1` and `hello2` definitions with these:

```
#[test]
fn hello1() -> TestResult {
    run(&["Hello there"], "tests/expected/hello1.txt") ❶
}

#[test]
fn hello2() -> TestResult {
    run(&["Hello", "there"], "tests/expected/hello2.txt") ❷
}

#[test]
fn hello1_no_newline() -> TestResult {
    run(&["Hello  there", "-n"], "tests/expected/hello1.n.txt") ❸
}

#[test]
fn hello2_no_newline() -> TestResult {
    run(&["-n", "Hello", "there"], "tests/expected/hello2.n.txt")
❹
}
```

❶ A single string value as input.

❷ Two strings as input.

❸ A single string value as input with `-n` flag to omit the newline. Note that there are two spaces between the words.

❹ Two strings as input with `-n` flag appearing first.

As you can see, I can write as many functions as I like in *tests/cli.rs*. Only those marked with `#[test]` are run when testing. If you run **`cargo test`** now, you should see five passing tests (in no particular order):

```
running 5 tests
test dies_no_args ... ok
test hello1 ... ok
test hello1_no_newline ... ok
test hello2_no_newline ... ok
test hello2 ... ok
```

Summary

Now you have written about 30 lines of Rust code in *src/main.rs* for the `echor` program and five tests in *tests/cli.rs* to verify that your program meets some measure of specification (the *specs*, as they say). Consider what you've achieved:

- Well-behaved systems programs should printed basic output to `STDOUT` and errors to `STDERR`.
- You've written a program that takes the options `-h|--help` to produce help, `-V|--version` to show the program's version, and `-n` to omit a newline along with one or more positional command line arguments.
- If the program is run with the wrong arguments or with the `-h|--help` flag, it will print usage documentation.
- The program will echo back all the command line arguments joined on spaces.
- The trailing newline will be omitted if the `-n` flag is present.
- You can run integration tests to confirm that your program replicates the output from `echo` for at least four test cases covering one or two inputs both with and without the trailing newline.
- You learned to use several Rust types including the unit type, strings, vectors, slices, `Option`, and `Result` as well as how to create type alias to a specific type of `Result` called a `TestResult`.
- You used a `Box` to create a smart pointer to heap memory. This required digging a bit into the differences between the stack—where variables have a fixed, known size and are accessed in LIFO order—and the heap—where the size of variables may change during the program and are accessed through a pointer.

- You learned how to read the entire contents of a file into a string.
- You learned how to execute an external command from within a Rust program, check the exit status, and verify the contents of both `STDOUT` and `STDERR`.

All this, and you've done it while writing in a language that simply will not allow you to make common mistakes that lead to buggy programs or security vulnerabilities. Feel free to give yourself a little high five or enjoy a slightly evil MWUHAHA chuckle as you consider how Rust will help you conquer the world. Now that I've shown how to organize and write tests and data, I'll use the tests earlier in the next program so I can start using *test-driven development* where I write tests first then write code to satisfy the tests.

---

1 "Trust, but verify." Apparently this rhymes in Russian and so sounds cooler than when Reagan used it in the 1980s during nuclear disarmament talks with the USSR.

Chapter 3. On The Catwalk

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 3rd Chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the author at kyclark@gmail.com.

When you are alone, you are the cat, you are the phone. You are an animal.

—They Might Be Giants

In this chapter, the challenge is to write a clone of the `cat` program, which is so named because it can *con**cat**enate* many files into one file. That is, given files *a*, *b*, and *c*, you could execute `cat a b c > all` to stream all the lines from these three files and redirect them into a file called *all*. The program will accept an option to prefix each line with the line number.

In this chapter, you'll learn:

- How to organize your code into a library and a binary crate
- How to use testing-first development
- The difference between public and private variables and functions
- How to test for the existence of a file

- How to create a random string for a file that does not exist
- How to read regular files or `STDIN` (pronounced *standard in*)
- How to use `eprintln!` to print to `STDERR` and `format!` to format a string
- How to write a test that provides input on `STDIN`
- How and why to create a `struct`
- How to define mutually exclusive arguments
- How to use the `enumerate` method of an iterator
- More about how and why to use a `Box`

How cat Works

I'll start by showing how `cat` works so that you know what is expected of the challenge. The BSD version of `cat` does not respond to `--help`, so I must use **`man cat`** to read the manual page. For such a limited program, it has a surprising number of options:

```
CAT(1)                    BSD General Commands Manual
CAT(1)

NAME
     cat -- concatenate and print files

SYNOPSIS
     cat [-benstuv] [file ...]

DESCRIPTION
     The cat utility reads files sequentially, writing them to the
standard
     output.  The file operands are processed in command-line
order.  If file
     is a single dash ('-') or absent, cat reads from the standard
input.  If
     file is a UNIX domain socket, cat connects to it and then
reads it until
```

```
     EOF.  This complements the UNIX domain binding capability
available in
     inetd(8).

     The options are as follows:

     -b      Number the non-blank output lines, starting at 1.

     -e      Display non-printing characters (see the -v option),
and display
             a dollar sign ('$') at the end of each line.

     -n      Number the output lines, starting at 1.

     -s      Squeeze multiple adjacent empty lines, causing the
output to be
             single spaced.

     -t      Display non-printing characters (see the -v option),
and display
             tab characters as '^I'.

     -u      Disable output buffering.

     -v      Display non-printing characters so they are visible.
Control
             characters print as '^X' for control-X; the delete
character
             (octal 0177) prints as '^?'.  Non-ASCII characters
(with the high
             bit set) are printed as 'M-' (for meta) followed by
the character
             for the low 7 bits.

EXIT STATUS
     The cat utility exits 0 on success, and >0 if an error occurs.
```

The GNU version does respond to `--help`:

```
$ cat --help
Usage: cat [OPTION]... [FILE]...
Concatenate FILE(s), or standard input, to standard output.

  -A, --show-all           equivalent to -vET
  -b, --number-nonblank    number nonempty output lines, overrides
-n
```

```
  -e                        equivalent to -vE
  -E, --show-ends           display $ at end of each line
  -n, --number              number all output lines
  -s, --squeeze-blank       suppress repeated empty output lines
  -t                        equivalent to -vT
  -T, --show-tabs           display TAB characters as ^I
  -u                        (ignored)
  -v, --show-nonprinting    use ^ and M- notation, except for LFD
and TAB
      --help     display this help and exit
      --version  output version information and exit

With no FILE, or when FILE is -, read standard input.

Examples:
  cat f - g  Output f's contents, then standard input, then g's
contents.
  cat        Copy standard input to standard output.

GNU coreutils online help: <http://www.gnu.org/software/coreutils/>
For complete documentation, run: info coreutils 'cat invocation'
```

NOTE

The BSD version predates the GNU version, so the latter implements all the same short flags to be compatible. As is typical of GNU programs, it also offers long flag aliases like `--number` for `-n` and `--number-nonblank` for `-b`. I will show you how to offer both options like the GNU version.

For the challenge program, I will only implement the options `-b|--number-nonblank` and `-n|--number`. I will also show how to read regular files and `STDIN` when given a filename argument of "-". I've put four files for testing into the *03_catr/tests/inputs* directory:

1. *empty.txt*: an empty file
2. *fox.txt*: a single line of text
3. *spiders.txt*: three lines of text
4. *the-bustle.txt*: a lovely poem by Emily Dickinson that has nine lines including one blank

Empty files are common, if useless. I include this to ensure my program can gracefully handle unexpected input. That is, I want my program to at least not fall over. The following command produces no output, so I expect my program to do the same:

```
$ cd 03_catr
$ cat tests/inputs/empty.txt
```

Next, I'll run `cat` on a file with one line of text:

```
$ cat tests/inputs/fox.txt
The quick brown fox jumps over the lazy dog.
```

The `-n|--number` and `-b|--number-nonblank` flags will both number the lines, and the line number is right-justified in a field six characters wide followed by a tab character and then the line of text. To distinguish the tab character, I can use the `-t` option to display non-printing characters so that the tab shows as `^I`. In the following command, I use the Unix pipe `|` to connect `STDOUT` from the first command to `STDIN` in the second command:

```
$ cat -n tests/inputs/fox.txt | cat -t
     1^IThe quick brown fox jumps over the lazy dog.
```

The *spiders.txt* file has three lines of text which should be numbered with the `-b` option:

```
$ cat -b tests/inputs/spiders.txt
     1  Don't worry, spiders,
     2  I keep house
     3  casually.
```

The difference between `-n` (on the left) and `-b` (on the right) is apparent only with *the-bustle.txt* as the latter will only number nonblank lines:

```
$ cat -n tests/inputs/the-bustle.txt    $ cat -b tests/inputs/the-
bustle.txt
```

```
     1  The bustle in a house                1  The bustle in a
house
     2  The morning after death              2  The morning after
death
     3  Is solemnest of industries           3  Is solemnest of
industries
     4  Enacted upon earth,—                 4  Enacted upon earth,
—
     5
     6  The sweeping up the heart,           5  The sweeping up the
heart,
     7  And putting love away                6  And putting love
away
     8  We shall not want to use again       7  We shall not want
to use again
     9  Until eternity.                      8  Until eternity.
```

> **NOTE**
>
> Oddly, you can use `-b` and `-n` together, and the `-b` option takes precedence. The challenge program will allow only one or the other.

When processing any file that does not exist or cannot be opened, `cat` will print a message to STDERR and move to the next file. In the following example, I'm using *blargh* as a nonexistent file. I create the file *cant-touch-this* using the `touch` command and use the `chmod` command to set the file with the permissions that make it unreadable. You'll learn more about what the `000` means in Chapter 15 when you write a clone of `ls`:

```
$ touch cant-touch-this && chmod 000 cant-touch-this
$ cat tests/inputs/fox.txt blargh tests/inputs/spiders.txt cant-
touch-this
The quick brown fox jumps over the lazy dog. ❶
cat: blargh: No such file or directory ❷
Don't worry, spiders, ❸
I keep house
casually.
cat: cant-touch-this: Permission denied ❹
```

❶ This is the output from the first file.

❷ This is an error for a nonexistent file.

❸ This is the output from the third file.

❹ This is the error for an unreadable file.

Finally, run `cat` with all the files and notice that it starts renumbering the lines for each file:

```
$ cd tests/inputs
$ cat -n empty.txt fox.txt spiders.txt the-bustle.txt
     1  The quick brown fox jumps over the lazy dog.
     1  Don't worry, spiders,
     2  I keep house
     3  casually.
     1  The bustle in a house
     2  The morning after death
     3  Is solemnest of industries
     4  Enacted upon earth,—
     5
     6  The sweeping up the heart,
     7  And putting love away
     8  We shall not want to use again
     9  Until eternity.
```

If you look at the *mk-outs.sh* script, you'll see I execute `cat` with all these files, individually and together, as regular files and through `STDIN`, using no flags and with the `-n` and `-b` flags. I capture all the outputs to various files in the *tests/expected* directory to use in testing.

Getting Started with Test-Driven Development

In Chapter 2, I wrote the tests at the end of the chapter because I needed to show you some basics of the language. Starting with this exercise, I want to make you think about *test-driven development* (TDD) as described in a book by that title written by Kent Beck (Addison-Wesley, 2002). TDD advocates writing tests for code *before* writing the code as shown in Figure 3-1.

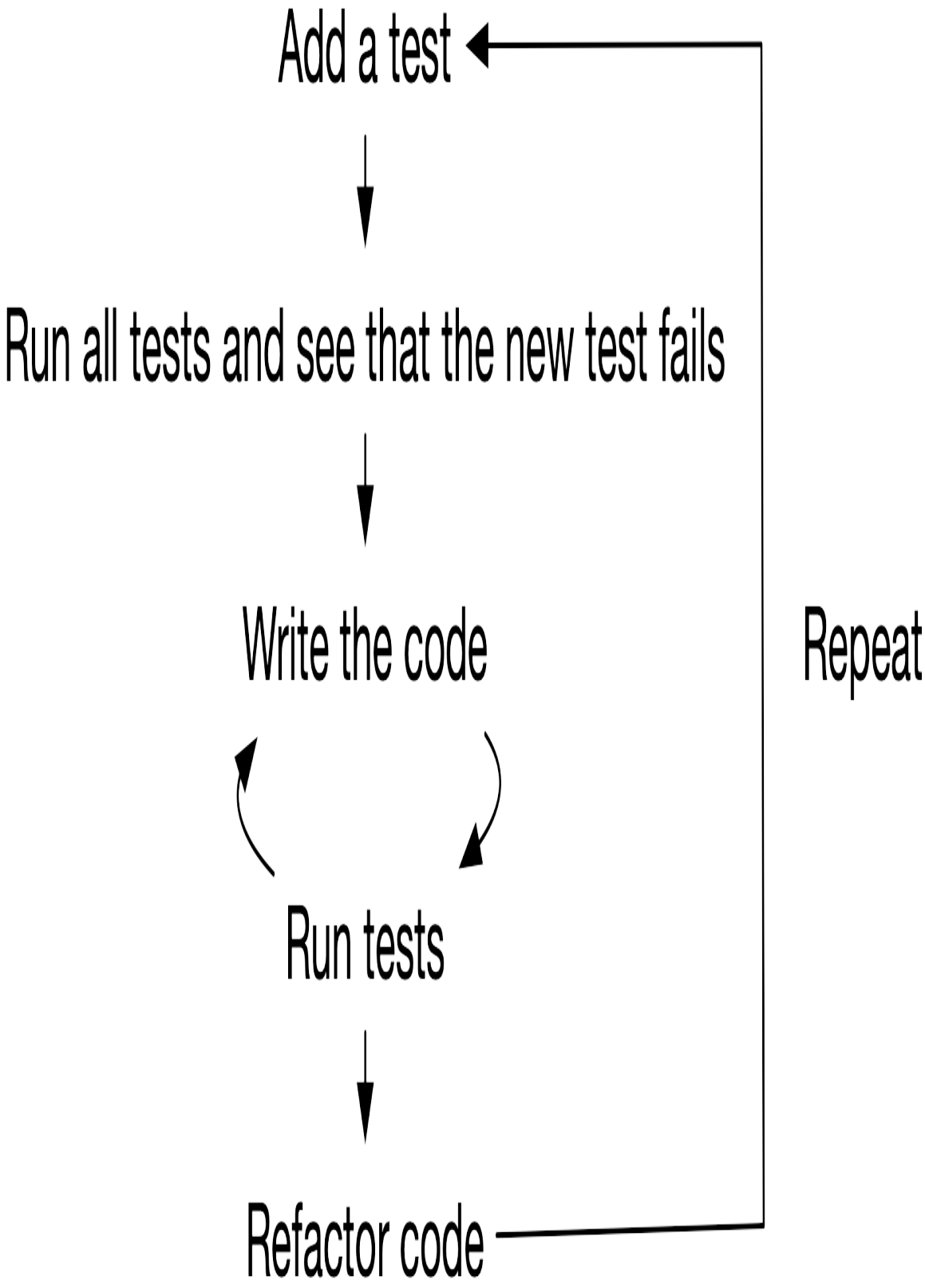

Add a test
Run all tests and see that the new test fails
Write the code
Run tests
Refactor code
Repeat

Figure 3-1. The test-driven development cycle

> **NOTE**
>
> Technically, TDD involves writing each test as you add a feature. Since I've written all the tests for you, you might consider this more like *test-first development*. Once you've written code that passes the tests, you can start to refactor your code to improve it, perhaps by shortening the lines of code or by finding a faster implementation.

The challenge program you write should be called `catr` (pronounced *cat-er*) for a Rust version of `cat`. I suggest you begin with **`cargo new catr`** to start a new application and then copy my *03_catr/tests* directory into your source tree. Don't copy anything but the tests as you will write the rest of the code yourself. You should have a structure like this:

```
$ tree -L 2 catr/
catr
├── Cargo.toml
├── src
│   └── main.rs
└── tests
    ├── cli.rs
    ├── expected
    └── inputs

4 directories, 3 files
```

I'm going to use all the same external crates as in Chapter 2 plus the `rand` crate for testing, so update your *Cargo.toml* to this:

```
[dependencies]
clap = "2.33"

[dev-dependencies]
assert_cmd = "1"
predicates = "1"
rand = "0.8"
```

Now run **`cargo test`** to download the crates, compile your program, and

run the tests. All the tests should fail. Your mission, should you choose to accept it, is to write a program that will pass these tests.

Creating a Library Crate

The program in Chapter 2 was quite short and easily fit into *src/main.rs*. The typical programs you will write in your career will likely be much longer. Starting with this program, I will divide my code into a library in *src/lib.rs* and a binary in *src/main.rs* that will call library functions. I believe this organization makes it easier to test and grow applications over time.

First, I'll move all the important bits from *src/main.rs* into a function called `run` in *src/lib.rs*. This function will return a kind of `Result` to indicate success or failure. This is similar to the `TestResult` type alias from Chapter 2. The `TestResult` always returns the unit type `()` in the `Ok` variant, but `MyResult` can return an `Ok` that contains any type which I can denote using the generic `T`:

```
use std::error::Error; ❶

type MyResult<T> = Result<T, Box<dyn Error>>;❷

pub fn run() -> MyResult<()> { ❸
    println!("Hello, world!"); ❹
    Ok(()) ❺
}
```

❶ Import the `Error` trait for representing error values.

❷ Create a `MyResult` to represent an `Ok` value for any type `T` or some `Err` value that implements the `Error` trait.

❸ Define a *public* (`pub`) function that returns either `Ok` containing the unit type `()` or some error `Err`.

❹ Print *Hello, world!*.

❺ Return an indication that the function ran successfully.

> **TIP**
>
> By default, all the variables and functions in a module are private. In the preceding code, I must use `pub` to make this library function accessible to the rest of the program.

To call this, change *src/main.rs* to this:

```
fn main() {
    if let Err(e) = catr::run() { ❶
        eprintln!("{}", e); ❷
        std::process::exit(1); ❸
    }
}
```

❶ Execute the `catr::run` function and check if the return value matches an `Err(e)` where `e` is some value that implements the `Error` trait, which means among other things that it can be printed.

❷ Use the `eprintln!` (*error print line*) macro to print the error message `e` to `STDERR`.

❸ Exit the program with a nonzero value to indicate an error.

> **TIP**
>
> The `eprint!` and `eprintln!` macros are just like `print!` and `println!` except that they print to `STDERR`.

If you execute **`cargo run`**, you should see *Hello, world!* as before.

Defining the Parameters

Next, I'll add the program's parameters, and I'd like to introduce a `struct` called `Config` to represent the arguments to the program. A `struct` is a data structure in which you define the names and types of the elements it will contain. It's similar to a class definition in other languages. In this case, I

want a `struct` that describes the values the program will need such as a list of the input filenames and the flags for numbering the lines of output.

Add the following `struct` to *src/lib.rs*. It's common to place such definitions near the top after the `use` statements:

```
#[derive(Debug)] ❶
pub struct Config { ❷
    files: Vec<String>, ❸
    number_lines: bool, ❹
    number_nonblank_lines: bool, ❺
}
```

❶ The `derive` macro allows me to add the `Debug` trait so the struct can be printed.

❷ Define a `struct` called `Config`. The `pub` (*public*) makes this accessible outside the library.

❸ The `files` will be a vector of strings.

❹ This is a Boolean value to indicate whether or not to print the line numbers.

❺ This is a Boolean to control printing line numbers only for nonblank lines.

To use a `struct`, I can create an instance of it with specific values. In the following sketch of a `get_args` function, you can see it finishes by creating a new `Config` with the runtime values from the user. Add `use clap::{App, Arg}` and this function to your *src/lib.rs*. Try to complete the function on your own, stealing what you can from Chapter 2:

```
pub fn get_args() -> MyResult<Config> { ❶
    let matches = App::new("catr")... ❷

    Ok(Config { ❸
        files: ...,
        number_lines: ...,
        number_nonblank_lines: ...,
    })
```

```
}
```

- ❶ This is a public function that returns a `MyResult` that will contain either a `Config` on success or an error.
- ❷ Here you should define the parameters and process the matches.
- ❸ Create a `Config` using the supplied values.

This means the `run` function needs to be updated to accept a `Config` argument. For now, print it:

```
pub fn run(config: Config) -> MyResult<()> { ❶
    dbg!(config); ❷
    Ok(())
}
```

- ❶ The function will accept a `Config` struct and will return `Ok` with the unit type if successful.
- ❷ Use the `dbg!` (*debug*) macro to print the configuration.

Update your *src/main.rs* as follows:

```
fn main() {
    if let Err(e) = catr::get_args().and_then(catr::run) { ❶
        eprintln!("{}", e); ❷
        std::process::exit(1); ❸
    }
}
```

- ❶ Call the `catr::get_args` function, then use `Result::and_then` to pass the `Ok(config)` to `catr::run`.
- ❷ If either `get_args` or `run` returns an `Err`, print it to `STDERR`.
- ❸ Exit the program with a nonzero value.

See if you can get your program to print a usage like this:

```
$ cargo run --quiet -- --help
catr 0.1.0
Ken Youens-Clark <kyclark@gmail.com>
Rust cat

USAGE:
    catr [FLAGS] <FILE>...

FLAGS:
    -h, --help               Prints help information
    -n, --number             Number lines
    -b, --number-nonblank    Number non-blank lines
    -V, --version            Prints version information

ARGS:
    <FILE>...    Input file(s) [default: -]
```

With no arguments, you program should be able to print a configuration structure like this:

```
$ cargo run
[src/lib.rs:52] config = Config {
    files: [ ❶
        "-",
    ],
    number_lines: false, ❷
    number_nonblank_lines: false,
}
```

❶ The default `files` should contain "-" for `STDIN`.

❷ The Boolean values should default to `false`.

Run with arguments and be sure the `config` looks like this:

```
$ cargo run -- -n tests/inputs/fox.txt
[src/lib.rs:52] config = Config {
    files: [
        "tests/inputs/fox.txt", ❶
    ],
    number_lines: true, ❷
    number_nonblank_lines: false,
}
```

❶ The positional file argument is parsed into the `files`.

❷ The `-n` option causes `number_lines` to be `true`.

While the BSD version will allow both `-n` and `-b`, the challenge program should consider these to be mutually exclusive and generate an error when used together:

```
$ cargo run -- -b -n tests/inputs/fox.txt
error: The argument '--number-nonblank' cannot be used with '--
number'
```

Give it a go. Seriously! I want you to try writing your version of this before you read ahead. I'll wait here until you finish.

All set? Compare what you have to my `get_args` function:

```
pub fn get_args() -> MyResult<Config> {
    let matches = App::new("catr")
        .version("0.1.0")
        .author("Ken Youens-Clark <kyclark@gmail.com>")
        .about("Rust cat")
        .arg(
            Arg::with_name("files") ❶
                .value_name("FILE")
                .help("Input file(s)")
                .required(true)
                .default_value("-")
                .min_values(1),
        )
        .arg(
            Arg::with_name("number") ❷
                .help("Number lines")
                .short("n")
                .long("number")
                .takes_value(false)
                .conflicts_with("number_nonblank"),
        )
        .arg(
            Arg::with_name("number_nonblank") ❸
                .help("Number non-blank lines")
                .short("b")
                .long("number-nonblank")
```

```
                .takes_value(false),
        )
        .get_matches();

    Ok(Config {
        files: matches.values_of_lossy("files").unwrap(), ❹
        number_lines: matches.is_present("number"), ❺
        number_nonblank_lines:
matches.is_present("number_nonblank"),
    })
}
```

❶ This positional argument is for the files and is required to have at least one value that defaults to "-".

❷ This is an option that has a `short` name `-n` and a `long` name `--number`. It does not take a value because it is a flag. When present, it will tell the program to print line numbers. It cannot occur in conjunction with `-b`.

❸ The `-b|--number-nonblank` flag controls whether to print line numbers for nonblank lines.

❹ Because at least one value is required, it should be safe to call `Option::unwrap`.

❺ The two Boolean options are either present or not.

> **TIP**
>
> Optional arguments have `short` and/or `long` names, but positional ones do not. You can define optional arguments before or after positional arguments. Defining positional arguments with `min_values` also implies `multiple` values but does not for optional parameters.

With this much code, you should be able to pass at least a couple of the tests when you execute **`cargo test`**. There will be a great deal of output showing you all the failing test output, but don't despair. You will soon see a fully passing test suite.

Processing the Files

Now that you have validated all the arguments, you are ready to process the files and create the correct output. First modify the `run` function in *src/lib.rs* to print each filename:

```
pub fn run(config: Config) -> MyResult<()> {
    for filename in config.files { ❶
        println!("{}", filename); ❷
    }
    Ok(())
}
```

❶ Iterate through each filename.

❷ Print the filename.

Run the program with some input files. In the following example, the `bash` shell will expand the file glob **.txt* into all filenames that end with the extension *.txt*:

```
$ cargo run -- tests/inputs/*.txt ❶
tests/inputs/empty.txt
tests/inputs/fox.txt
tests/inputs/spiders.txt
tests/inputs/the-bustle.txt
```

❶ TK

Windows PowerShell can expand file globs using `Get-ChildItem`:

```
> cargo run -q -- -n (Get-ChildItem .\tests\inputs\*.txt)
C:\Users\kyclark\work\rust-sysprog\03_catr\tests\inputs\empty.txt
C:\Users\kyclark\work\rust-sysprog\03_catr\tests\inputs\fox.txt
C:\Users\kyclark\work\rust-sysprog\03_catr\tests\inputs\spiders.txt
C:\Users\kyclark\work\rust-sysprog\03_catr\tests\inputs\the-
bustle.txt
```

Opening a File or STDIN

The next step is to try to open each filename. When the filename is "-", I should open `STDIN`; otherwise, I will attempt to open the given filename and handle errors. For the following code, you will need to expand your imports to the following:

```
use clap::{App, Arg};
use std::error::Error;
use std::fs::File;
use std::io::{self, BufRead, BufReader};
```

This next step is a bit tricky, so I'd like to provide an `open` function for you to use. In the following code, I'm using the `match` keyword, which is similar to a `switch` statement in C. Specifically, I'm matching on whether `filename` is equal to "-" or something else, which is specified using the wildcard _:

```
fn open(filename: &str) -> MyResult<Box<dyn BufRead>> { ❶
    match filename {
        "-" => Ok(Box::new(BufReader::new(io::stdin()))), ❷
        _ => Ok(Box::new(BufReader::new(File::open(filename)?))),
❸
    }
}
```

- ❶ The function will accept the filename and will return either an error or a boxed value that implements the `BufRead` trait.
- ❷ When the filename is "-", read from `std::io::stdin`.
- ❸ Otherwise, use `File::open` to try to open the given file or propagate an error.

If `File::open` is successful, the result will be a *filehandle*, which is a mechanism for reading the contents of a file. Both a filehandle and `std::io::stdin` implement the `BufRead` trait, which means the values will, for instance, respond to the `BufRead::lines` function to produce lines of text. Note that `BufRead::lines` will remove any line endings such as `\r\n` on Windows and `\n` on Unix.

Again you see I'm using a `Box` to create a pointer to heap-allocated memory to hold the filehandle. You may wonder if this is completely necessary. I could try to write the function without using `Box`:

```
// This will not compile
fn open(filename: &str) -> MyResult<dyn BufRead> {
    match filename {
        "-" => Ok(BufReader::new(io::stdin())),
        _ => Ok(BufReader::new(File::open(filename)?)),
    }
}
```

If I try to compile this code, I get the following error:

```
error[E0277]: the size for values of type `(dyn std::io::BufRead +
'static)`
cannot be known at compilation time
   --> src/lib.rs:88:28
    |
88  | fn open(filename: &str) -> MyResult<dyn BufRead> {
    |                            ^^^^^^^^^^^^^^^^^^^^
    |                            doesn't have a size known at
compile-time
    |
    = help: the trait `Sized` is not implemented for `(dyn
std::io::BufRead
    + 'static)`
```

As the compiler says, there is not an implementation in `BufRead` for the `Sized` trait. If a variable doesn't have a fixed, known size, then Rust can't store it on the stack. The solution is to instead allocate memory on the heap by putting the return value into a `Box`, which is a pointer with a known size.

The preceding `open` function is really dense. I can appreciate it if you think that's more than a little complicated; however, it handles basically any error you will encounter. To demonstrate this, change your `run` to the following:

```
pub fn run(config: Config) -> MyResult<()> {
    for filename in config.files { ❶
        match open(&filename) { ❷
            Err(err) => eprintln!("Failed to open {}: {}",
```

```
filename, err), ❸
            Ok(_) => println!("Opened {}", filename), ❹
        }
    }
    Ok(())
}
```

❶ Iterate through the filenames.

❷ Try to open the filename. Note the use of & to borrow the variable.

❸ Print an error message to `STDERR` when `open` fails.

❹ Print a successful message when `open` works.

Try to run your program with the following:

1. A valid input file
2. A nonexistent file
3. An unreadable file

For the last option, you can create a file that cannot be read like so:

```
$ touch cant-touch-this && chmod 000 cant-touch-this
```

Run your program and verify your code gracefully prints error messages for bad input files and continues to process the valid ones:

```
$ cargo run -- blargh cant-touch-this tests/inputs/fox.txt
Failed to open blargh: No such file or directory (os error 2)
Failed to open cant-touch-this: Permission denied (os error 13)
Opened tests/inputs/fox.txt
```

With this addition, you should be able to pass **`cargo test skips_bad_file`**. Now that you are able to open and read valid input files, I want you to finish the program on your own. Can you figure out how to read the opened file line-by-line? Start with *tests/inputs/fox.txt* that has only one line. You should be able to see the following output:

```
$ cargo run -- tests/inputs/fox.txt
The quick brown fox jumps over the lazy dog.
```

Verify that you can read `STDIN` by default. In the following command, I will use the `|` to pipe `STDOUT` from the first command to the `STDIN` of the second command:

```
$ cat tests/inputs/fox.txt | cargo run
The quick brown fox jumps over the lazy dog.
```

The output should be the same when providing the filename "-". In the following command, I will use the `bash` redirect operator `<` to take input from the given filename and provide it to `STDIN`:

```
$ cargo run -- - < tests/inputs/fox.txt
The quick brown fox jumps over the lazy dog.
```

Next, try an input file with more than one line and try to number the lines for `-n`:

```
$ cargo run -- -n tests/inputs/spiders.txt
     1  Don't worry, spiders,
     2  I keep house
     3  casually.
```

Then try to skip blank lines in the numbering for `-b`:

```
$ cargo run -- -b tests/inputs/the-bustle.txt
     1  The bustle in a house
     2  The morning after death
     3  Is solemnest of industries
     4  Enacted upon earth,—

     5  The sweeping up the heart,
     6  And putting love away
     7  We shall not want to use again
     8  Until eternity.
```

Run **`cargo test`** often to see which tests are failing. The tests in

tests/cli.rs are similar to Chapter 2, but I've added a little more organization. For instance, I define several constant `&str` values at the top of that module which I use throughout the crate. I use a common convention of `ALL_CAPS` names to highlight the fact that they are *scoped* or visible throughout the crate:

```
const PRG: &str = "catr";
const EMPTY: &str = "tests/inputs/empty.txt";
const FOX: &str = "tests/inputs/fox.txt";
const SPIDERS: &str = "tests/inputs/spiders.txt";
const BUSTLE: &str = "tests/inputs/the-bustle.txt";
```

To test that the program will die when given a nonexistent file, I use the `rand` crate to generate a random filename that does not exist. For the following function, I will `use rand::{distributions::Alphanumeric, Rng}` to import various parts of the crate I need in this function:

```
fn gen_bad_file() -> String { ❶
    loop { ❷
        let filename: String = rand::thread_rng() ❸
            .sample_iter(&Alphanumeric)
            .take(7)
            .map(char::from)
            .collect();

        if fs::metadata(&filename).is_err() { ❹
            return filename;
        }
    }
}
```

❶ The function will return a `String`, which is a dynamically generated string closely related to the `str` struct I've been using.

❷ Start an infinite `loop`.

❸ Create a random string of seven alphanumeric characters.

❹ `fs::metadata` returns an error when the given filename does not exist, so return the nonexistent filename.

> **NOTE**
>
> In the preceding function, I use `filename` two times after creating it. The first time, I borrow it using `&filename`, and the second time I don't use the ampersand. Try removing the & and running the code. You should get an error message that ownership of `filename` value is moved into `fs::metadata`. Effectively, the function consumes the value, leaving it unusable. The & shows I only want to borrow a reference to the value.

```
error[E0382]: use of moved value: `filename`
  --> tests/cli.rs:37:20
   |
30 |         let filename: String = rand::thread_rng()
   |             -------- move occurs because `filename` has type
`String`,
   |                      which does not implement the `Copy` trait
...
36 |         if fs::metadata(filename).is_err() {
   |                         -------- value moved here
37 |             return filename;
   |                    ^^^^^^^^ value used here after move
```

Don't worry if you don't completely understand the preceding code yet. I'm only showing this so you understand how it is used in the `skips_bad_file` test:

```
#[test]
fn skips_bad_file() -> TestResult {
    let bad = gen_bad_file(); ❶
    let expected = format!("{}: .* [(]os error 2[)]", bad); ❷
    Command::cargo_bin(PRG)? ❸
        .arg(&bad)
        .assert()
        .success() ❹
        .stderr(predicate::str::is_match(expected)?);
    Ok(())
}
```

❶ Generate the name of a nonexistent file.

❷ The expected error message should include the filename and the string "os error 2" on both Windows or Unix platforms.

❸ Run the program with the bad file and verify that `STDERR` matches the expected pattern.

❹ The command should succeed as bad files should only generate warnings and not kill the process.

> **TIP**
>
> In the preceding function, I used the `format!` macro to generate a new `String`. This macro works like `print!` except that it returns the value rather than printing it.

I created a `run` helper function to run the program with input arguments and verify that the output matches the text in the file generated by *mk-outs.sh*:

```
fn run(args: &[&str], expected_file: &str) -> TestResult { ❶
    let expected = fs::read_to_string(expected_file)?; ❷
    Command::cargo_bin(PRG)? ❸
        .args(args)
        .assert()
        .success()
        .stdout(expected);
    Ok(())
}
```

❶ The function accepts a slice of `&str` arguments and the filename with the expected output. The function returns a `TestResult`.

❷ Try to read the expected output file.

❸ Execute the program with the arguments and verify it runs successfully and produces the expected output.

I use this function like so:

```
#[test]
fn bustle() -> TestResult {
    run(&[BUSTLE], "tests/expected/the-bustle.txt.out") ❶
}
```

❶ Run the program with the `BUSTLE` input file and verify that the output matches the output produced by *mk-outs.sh*.

I also wrote a helper function to provide input via `STDIN`:

```
fn run_stdin(
    input_file: &str, ❶
    args: &[&str],
    expected_file: &str,
) -> TestResult {
    let input = fs::read_to_string(input_file)?; ❷
    let expected = fs::read_to_string(expected_file)?;
    Command::cargo_bin(PRG)? ❸
        .args(args)
        .write_stdin(input)
        .assert()
        .success()
        .stdout(expected);
    Ok(())
}
```

❶ The first argument is the filename containing the text that should be given to `STDIN`.

❷ Try to read the input and expected files.

❸ Try to run the program with the given arguments and `STDIN` and verify the output.

This function is used similarly:

```
#[test]
fn bustle_stdin() -> TestResult {
    run_stdin(BUSTLE, &["-"], "tests/expected/the-
bustle.txt.stdin.out") ❶
}
```

❶ Run the program using the contents of the given filename as `STDIN` and an input filename of "-". Verify the output matches the expected value.

That should be enough to get started. Off you go! Come back when you're

done.

Solution

I hope you found this an interesting and challenging program to write. It's important to tackle complicated programs one step at a time. I'll show you how I built my program in this way.

Reading the Lines in a File

I started with printing the lines of an open filehandle:

```
pub fn run(config: Config) -> MyResult<()> {
    for filename in config.files {
        match open(&filename) {
            Err(err) => eprintln!("{}: {}", filename, err), ❶
            Ok(file) => {
                for line_result in file.lines() { ❷
                    let line = line_result?; ❸
                    println!("{}", line); ❹
                }
            }
        }
    }
    Ok(())
}
```

❶ Print the filename and error when there is a problem opening a file.

❷ Iterate over each `line_result` value from `BufRead::lines`.

❸ Either unpack an `Ok` value from `line_result` or propagate an error.

❹ Print the line.

> **NOTE**
>
> When reading the lines from a file, you don't get the lines directly from the filehandle but instead get a `std::io::Result`, which is a "type is broadly used across `std::io` for any operation which may produce an error." Reading and writing files falls into the category of IO (input/output) which depends on external resources like the operating and file systems. While it's unlikely that reading a line from a filehandle will fail, the point is that it *could* fail.

If you run **`cargo test`**, you should pass about half of the tests, which is not bad for so few lines of code.

Printing Line Numbers

Next, I'd like to add the printing of line numbers for the `-n|--number` option. One solution that will likely be familiar to C programmers would be something like this:

```
pub fn run(config: Config) -> MyResult<()> {
    for filename in config.files {
        match open(&filename) {
            Err(err) => eprintln!("{}: {}", filename, err),
            Ok(file) => {
                let mut line_num = 0; ❶
                for line_result in file.lines() {
                    let line = line_result?;
                    line_num += 1; ❷

                    if config.number_lines { ❸
                        println!("{:>6}\t{}", line_num, line); ❹
                    } else {
                        println!("{}", line); ❺
                    }
                }
            }
        }
    }
    Ok(())
}
```

❶ Initialize a mutable counter variable to hold the line number.

❷ Add 1 to the line number.

❸ Check whether to print line numbers.

❹ If so, print the current line number in a right-justified field 6 characters wide followed by a tab character, and then the line of text.

❺ Otherwise, print the line.

Recall that all variables in Rust are immutable by default, so it's necessary to add `mut` to `line_num` as I intend to change it. The `+=` operator is a compound assignment that adds the righthand value 1 to `line_num` to increment it[1]. Of note, too, is the formatting syntax `{:>6}` that indicates the width of the field as six characters with the text aligned to the right. (You can use `<` for left-justified and `^` for centered text.) This syntax is similar to `printf` in C, Perl, and Python's string formatting.

If I run this version of the program, it looks pretty good:

```
$ cargo run -- tests/inputs/spiders.txt -n
     1  Don't worry, spiders,
     2  I keep house
     3  casually.
```

While this works adequately, I'd like to point out a more idiomatic solution using `Iterator::enumerate`. This method will return a tuple containing the index position and value for each element in an *iterable*, which is something that can produce values until exhausted:

```
pub fn run(config: Config) -> MyResult<()> {
    for filename in config.files {
        match open(&filename) {
            Err(err) => eprintln!("{}: {}", filename, err),
            Ok(file) => {
                for (line_num, line_result) in
file.lines().enumerate() { ❶
                    let line = line_result?;
                    if config.number_lines {
                        println!("{:>6}\t{}", line_num + 1, line);
❷
                    } else {
                        println!("{}", line);
                    }
                }
            }
        }
    }
    Ok(())
}
```

❶ The tuple values from `Iterator::enumerate` can be unpacked using pattern matching.

❷ Numbering from `enumerate` starts at 0, so add 1 to mimic `cat` which starts at 1.

This will create the same output, but now the code avoids using a mutable value. I can execute **`cargo test fox`** to run all the tests starting with *fox*, and I find that two out of three pass. The program fails on the `-b` flag, so next I need to handle printing the line numbers only for nonblank lines. Notice in this version, I'm also going to remove `line_result` and shadow the `line` variable:

```
pub fn run(config: Config) -> MyResult<()> {
    for filename in config.files {
        match open(&filename) {
            Err(err) => eprintln!("{}: {}", filename, err),
            Ok(file) => {
                let mut last_num = 0; ❶
                for (line_num, line) in file.lines().enumerate() {
                    let line = line?; ❷
                    if config.number_lines { ❸
                        println!("{:>6}\t{}", line_num + 1, line);
                    } else if config.number_nonblank_lines { ❹
                        if !line.is_empty() {
                            last_num += 1;
                            println!("{:>6}\t{}", last_num, line);
❺
                        } else {
                            println!(); ❻
                        }
                    } else {
                        println!("{}", line); ❼
                    }
                }
            }
        }
    }
    Ok(())
}
```

❶ Initialize a mutable variable for the number of the last nonblank line.

❷ Shadow the `line` with the result of unpacking the `Result`.

❸ Handle printing line numbers.

❹ Handle printing line numbers for nonblank lines.

❺ If the line is not empty, increment `last_num` and print the output.

❻ If the line is empty, print a blank line.

❼ If there are no numbering options, print the line.

> **NOTE**
>
> *Shadowing* a variable is Rust is when you reuse a variable's name and set it to a new value. Arguably the `line_result`/`line` code may be more explicit and readable, but reusing `line` in this context is more Rustic code you're likely to encounter.

If you run **`cargo test`**, you should pass all the tests.

Going Further

You have a working program now, but you don't have to stop there. If you're up for an additional challenge, try implementing the other options shown in the manual pages for both the BSD and GNU versions. For each option, use `cat` to create the expected output file, then expand the tests to check that your program creates this same output. I'd also recommend you check out the `bat`, which is another Rust clone of `cat` ("with wings") for a more complete implementation.

The number lines output of `cat -n` is similar in ways to `nl`, a "line numbering filter" program. `cat` is also a bit similar to programs that will show you a *page* or screenfull of text at a time, so called *pagers* like `more` and `less`. (`more` would show you a page of text with "More" at the bottom to let you know you could continue. Obviously someone decided to be clever and named their clone `less`, but it does the same thing.) Consider implementing both of those programs. Read the manual pages, create the test

output, and copy the ideas from this project to write and test your versions.

Summary

You made big strides in this chapter, creating a much more complex program. Consider what you learned:

- You separated your code into library (*src/lib.rs*) and binary (*src/main.rs*) crates, which can make it easier to organize and encapsulate ideas.
- You created your first `struct`, which is a bit like a class declaration in other languages. This struct allowed you to create a complex data structure called `Config` to describe the inputs for your program.
- By default, all values and functions are immutable and private. You learned to use `mut` to make a value mutable and `pub` to make a value or function public.
- You used testing-first approach where all the tests exist before the program is even written. When the program passes all the tests, you can be confident your program meets all the specifications encoded in the tests.
- You saw how to use the `rand` crate to generate a random string for a nonexistent file.
- You figured out how to read lines of text from both `STDIN` or regular files.
- You used the `eprintln!` macros to print to `STDERR` and `format!` to dynamically generate a new string.
- You used a `for` loop to visit each element in an iterable.
- You found that the `Iterator::enumerate` method will return both the index and element as a tuple, which was useful for

numbering the lines of text.

- You learned to use a `Box` that points to a filehandle to read either `STDIN` or a regular file.

In the next chapter, you'll learn a good deal more about reading files by lines, bytes, or characters.

---

1 Note that Rust does not have a unary `++` operator, so you cannot use `line_num++` to increment a variable by 1.

Chapter 4. Head Aches

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 4th Chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the author at kyclark@gmail.com.

Stand on your own head for a change, give me some skin to call my own

—They Might Be Giants

The challenge in this chapter is to implement the `head` program, which will print the first few lines or bytes of one or more files. This is a good way to peek at the contents of a regular text file and is often a much better choice than `cat`. When faced with a directory of something like output files from some process, this is a great way to quickly scan for potential problems.

In this exercise, you will learn:

- How to create optional command-line arguments that accept values
- How to parse a string into a number
- How to write and run a unit test
- How to use a guard with a `match` arm
- How to convert between types using `From`, `Into`, and `as`

- How to use `take` on an iterator or a filehandle
- How to preserve line endings while reading a filehandle
- How to read bytes from a filehandle

How head Works

You should keep in mind that there are many implementations of the original AT&T Unix operating system, such as BSD (Berkeley Standard Distribution), SunOS/Solaris, HP-UX, and Linux. Most of these operating systems have some version of a `head` program that will default to showing the first ten lines of one or more files. Most will probably have options `-n` to control the number of lines shown and `-c` to instead show some number of bytes. The BSD version has only these two options, which I can see via **`man head`**:

```
HEAD(1)                   BSD General Commands Manual
HEAD(1)

NAME
     head -- display first lines of a file

SYNOPSIS
     head [-n count | -c bytes] [file ...]

DESCRIPTION
     This filter displays the first count lines or bytes of each of
the speci-
     fied files, or of the standard input if no files are
specified.  If count
     is omitted it defaults to 10.

     If more than a single file is specified, each file is preceded
by a
     header consisting of the string ''==> XXX <=='' where ''XXX''
is the name
     of the file.

EXIT STATUS
     The head utility exits 0 on success, and >0 if an error
```

```
occurs.

SEE ALSO
     tail(1)

HISTORY
     The head command appeared in PWB UNIX.

BSD                                 June 6, 1993
BSD
```

With the GNU version, I can run **`head --help`** to read the usage:

```
Usage: head [OPTION]... [FILE]...
Print the first 10 lines of each FILE to standard output.
With more than one FILE, precede each with a header giving the file
name.
With no FILE, or when FILE is -, read standard input.

Mandatory arguments to long options are mandatory for short options
too.
  -c, --bytes=[-]K         print the first K bytes of each file;
                             with the leading '-', print all but
the last
                             K bytes of each file
  -n, --lines=[-]K         print the first K lines instead of the
first 10;
                             with the leading '-', print all but
the last
                             K lines of each file
  -q, --quiet, --silent    never print headers giving file names
  -v, --verbose            always print headers giving file names
      --help     display this help and exit
      --version  output version information and exit

K may have a multiplier suffix:
b 512, kB 1000, K 1024, MB 1000*1000, M 1024*1024,
GB 1000*1000*1000, G 1024*1024*1024, and so on for T, P, E, Z, Y.
```

Note the ability with the GNU version to specify `-n` and `-c` with negative numbers and using suffixes like *K*, *M*, etc., which I will not implement. In both versions, the files are optional positional arguments that will read `STDIN` by default or when a filename is "-". The `-n` and `-b` are optional arguments that take integer values.

To demonstrate some examples using `head`, I'll use the files found in *04_headr/tests/inputs*. Given an empty file, there is no output, which you can verify with **`head tests/inputs/empty.txt`**. By default, `head` will print the first 10 lines of a file. If a file has fewer than 10 lines, it will print all the lines. You can see this using *tests/inputs/three.txt*, which has 3 lines:

```
$ cd 04_headr
$ head tests/inputs/three.txt
Three
lines,
four words.
```

The `-n` option allows you to control how many lines are shown. For instance, I can choose only 2 lines with the following command:

```
$ head -n 2 tests/inputs/three.txt
Three
lines,
```

The `-c` option shows only the given number of bytes from a file, for instance, just the first 4 bytes:

```
$ head -c 4 tests/inputs/three.txt
Thre
```

Oddly, the GNU version will allow you to provide both `-n` and `-c` and defaults to showing bytes. The BSD version will reject both arguments:

```
$ head -n 1 -c 2 tests/inputs/one.txt
head: can't combine line and byte counts
```

Any value for `-n` or `-c` that is not a positive integer will generate an error that will halt the program, and the error will echo back the illegal value:

```
$ head -n 0 tests/inputs/one.txt
head: illegal line count -- 0
$ head -c foo tests/inputs/one.txt
head: illegal byte count -- foo
```

When there are multiple arguments, `head` adds a header and inserts a blank line between each file:

```
$ head -n 1 tests/inputs/*.txt
==> tests/inputs/empty.txt <==

==> tests/inputs/one.txt <==
Öne line, four words.

==> tests/inputs/three.txt <==
Three

==> tests/inputs/two.txt <==
Two lines.
```

With no file arguments, `head` will read from `STDIN`:

```
$ cat tests/inputs/three.txt | head -n 2
Three
lines,
```

As with `cat` in Chapter 3, any nonexistent or unreadable file is skipped with a warning printed to `STDERR`. In the following command, I will use *blargh* as a nonexistent file and will create an unreadable file called *cant-touch-this*:

```
$ touch cant-touch-this && chmod 000 cant-touch-this
$ head blargh cant-touch-this tests/inputs/one.txt
head: blargh: No such file or directory
head: cant-touch-this: Permission denied
==> tests/inputs/one.txt <==
Öne line, four words.
```

This will be as much as the challenge program is expected to recreate.

Getting Started

You might have anticipated that the program I want you to write will be called `headr` (pronounced *head-er*). Start by running **`cargo new headr`** and copy my *04_headr/tests* directory into your project directory. Add the

following dependencies to your *Cargo.toml*:

```
[dependencies]
clap = "2.33"

[dev-dependencies]
assert_cmd = "1"
predicates = "1"
rand = "0.8"
```

I propose you again split your source code so that *src/main.rs* looks like this:

```
fn main() {
    if let Err(e) = headr::get_args().and_then(headr::run) {
        eprintln!("{}", e);
        std::process::exit(1);
    }
}
```

Begin your *src/lib.rs* by bringing in `clap` and the `Error` trait and declaring `MyResult`, which you can copy from the source code in Chapter 3:

```
use clap::{App, Arg};
use std::error::Error;

type MyResult<T> = Result<T, Box<dyn Error>>;
```

The program will have three parameters that can be represented with a `Config` struct:

```
#[derive(Debug)]
pub struct Config {
    files: Vec<String>, ❶
    lines: usize, ❷
    bytes: Option<usize>, ❸
}
```

❶ The `files` will be a vector of strings.

❷ The number of `lines` to print will be of the type `usize`.

❸ The `bytes` will be an optional `usize`.

The primitive `usize` is the "pointer-sized unsigned integer type," and its size varies from 4 bytes on a 32-bit operating system to 8 bytes on a 64-bit. The choice of `usize` is somewhat arbitrary as I just want to store some sort of positive integer. I could also use a `u32` (unsigned 32-bit integer) or a `u64` (unsigned 64-bit integer), but I definitely want an *unsigned* type as it will only represent positive integer values. I would need to use a *signed* integer like `i32` or `i64` to represent positive or negative numbers, which would be needed if I wanted to allow negative values as the GNU version does.

The `lines` and `bytes` will be used in a couple of functions, one of which expects a `usize` and the other `u64`. This will provide an opportunity later to discuss how to convert between types. Your program should use `10` as the default value for `lines`, but the `bytes` will be an `Option`, which I first introduced in Chapter 2. This means that `bytes` will either be `Some<usize>` if the user provides a valid value or `None` if they do not.

You can start your `get_args` function with the following outline. You need to add the code to parse the arguments and return a `Config` struct:

```
pub fn get_args() -> MyResult<Config> {
    let matches = App::new("headr")
        .version("0.1.0")
        .author("Ken Youens-Clark <kyclark@gmail.com>")
        .about("Rust head")
        ... // what goes here?
        .get_matches();

    Ok(Config {
        files: ...
        lines: ...
        bytes: ...
    })
}
```

TIP

All the command-line arguments for this program are optional because `files` will default to "-", `lines` will default to 10, and `bytes` can be left out. The optional arguments in Chapter 3 were flags, but here `lines` and `bytes` will need `Arg::takes_value` set to `true`.

You can start off with a `run` function that prints the configuration:

```
pub fn run(config: Config) -> MyResult<()> {
    println!("{:#?}", config); ❶
    Ok(()) ❷
}
```

❶ Pretty-print the `config`. You could also use `dbg!(config)`.

❷ Return a successful result.

Parsing Strings into Numbers

All the values that `clap` returns will be strings, but you will need to convert `lines` and `bytes` to integers when present. I will show you how to use `str::parse` for this. This function will return a `Result` that will be an `Err` when the provided value cannot be parsed into a number or an `Ok` containing the converted number. I will write a function called `parse_positive_int` that attempts to parse a string value into a positive `usize` value. You can add this to your *src/lib.rs*:

```
fn parse_positive_int(val: &str) -> MyResult<usize> { ❶
    unimplemented!(); ❷
}
```

❶ This function accepts a `&str` and will either return a positive `usize` or an error.

❷ The `unimplemented!` macro "indicates unimplemented code by panicking with a message of *not implemented*."

In the spirit of test-driven development, I will add a *unit* test for this function. I would recommend adding this just after the function it's testing:

```
#[test]
fn test_parse_positive_int() {
    // 3 is an OK integer
```

```
    let res = parse_positive_int("3");
    assert!(res.is_ok());
    assert_eq!(res.unwrap(), 3);

    // Any string is an error
    let res = parse_positive_int("foo");
    assert!(res.is_err());
    assert_eq!(res.unwrap_err().to_string(), "foo".to_string());

    // A zero is an error
    let res = parse_positive_int("0");
    assert!(res.is_err());
    assert_eq!(res.unwrap_err().to_string(), "0".to_string());
}
```

Run **`cargo test parse_positive_int`** and verify that, indeed, the test fails. Stop reading now and write a version of the function that passes this test. I'll wait here until you finish.

```
TIME PASSES.
AUTHOR GETS A CUP OF TEA AND CONSIDERS HIS LIFE CHOICES.
AUTHOR RETURNS TO THE NARATIVE.
```

How did that go? Swell, I bet! Here is the function I wrote that passes the preceding tests:

```
fn parse_positive_int(val: &str) -> MyResult<usize> {
    match val.parse() { ❶
        Ok(n) if n > 0 => Ok(n), ❷
        _ => Err(From::from(val)), ❸
    }
}
```

❶ Attempt to parse the given value. Rust infers the `usize` type from the return type.

❷ If the parse succeeds and the parsed value `n` is greater than 0, return it as an `Ok` variant.

❸ For any other outcome, return an `Err` with the given value.

I've used `match` several times so far, but this is the first time I'm showing

that `match` arms can include a *guard*, which is an additional check after the pattern match. I don't know about you, but I think that's pretty sweet.

Converting Strings into Errors

When I'm unable to parse a given string value into a positive integer, I want to return the original string so it can be included in an error message. To do this in the preceding function, I used the redundantly named `From::from` function to turn the input `&str` value into an `Error`. Consider this version where I try to put the unparsable string directly into the `Err`:

```
fn parse_positive_int(val: &str) -> MyResult<usize> {
    match val.parse() {
        Ok(n) if n > 0 => Ok(n),
        _ => Err(val), // This will not compile
    }
}
```

If I try to compile this, I get the following error:

```
error[E0308]: mismatched types
  --> src/lib.rs:75:18
   |
75 |         _ => Err(val), // This will not compile
   |                  ^^^
   |                  |
   |                  expected struct `Box`, found `&str`
   |                  help: store this in the heap by calling
`Box::new`:
   |                  `Box::new(val)`
   |
   = note: expected struct `Box<dyn std::error::Error>`
           found reference `&str`
   = note: for more on the distinction between the stack and the
heap,
   read https://doc.rust-lang.org/book/ch15-01-box.html,
   https://doc.rust-lang.org/rust-by-example/std/box.html,
   and https://doc.rust-lang.org/std/boxed/index.html
```

The problem is that I am expected to return a `MyResult` which is defined as either an `Ok<T>` for any kind of type `T` or something that implements the

`Error` trait and which is stored in a `Box`:

```
type MyResult<T> = Result<T, Box<dyn Error>>;
```

In the preceding code, `&str` neither implements `Error` nor lives in a `Box`. I can try to fix this according to the suggestions by changing this to `Err(Box::new(val))`. Unfortunately, this still won't compile as I still haven't satisfied the `Error` trait:

```
error[E0277]: the trait bound `str: std::error::Error` is not
satisfied
  --> src/lib.rs:75:18
   |
75 |         _ => Err(Box::new(val)), // This will not compile
   |                  ^^^^^^^^^^^^^ the trait `std::error::Error`
is not
   |                                implemented for `str`
   |
   = note: required because of the requirements on the impl of
     `std::error::Error` for `&str`
   = note: required for the cast to the object type `dyn
std::error::Error`
```

Enter the `std::convert::From` trait, which helps convert from one type to another. For example, the documentation shows how to convert from a `str` to a `String`:

```
let string = "hello".to_string();
let other_string = String::from("hello");
assert_eq!(string, other_string);
```

In my case, I can convert `&str` into an `Error` in several ways using both `std::convert::From` and `std::convert::Into`. As the documentation states:

> *The `From` is also very useful when performing error handling. When constructing a function that is capable of failing, the return type will generally be of the form `Result<T, E>`. The `From` trait simplifies error handling by allowing a function to return a single error type that*

encapsulate multiple error types.

Figure 4-1 shows several equivalent ways to write this, none of which are preferable.

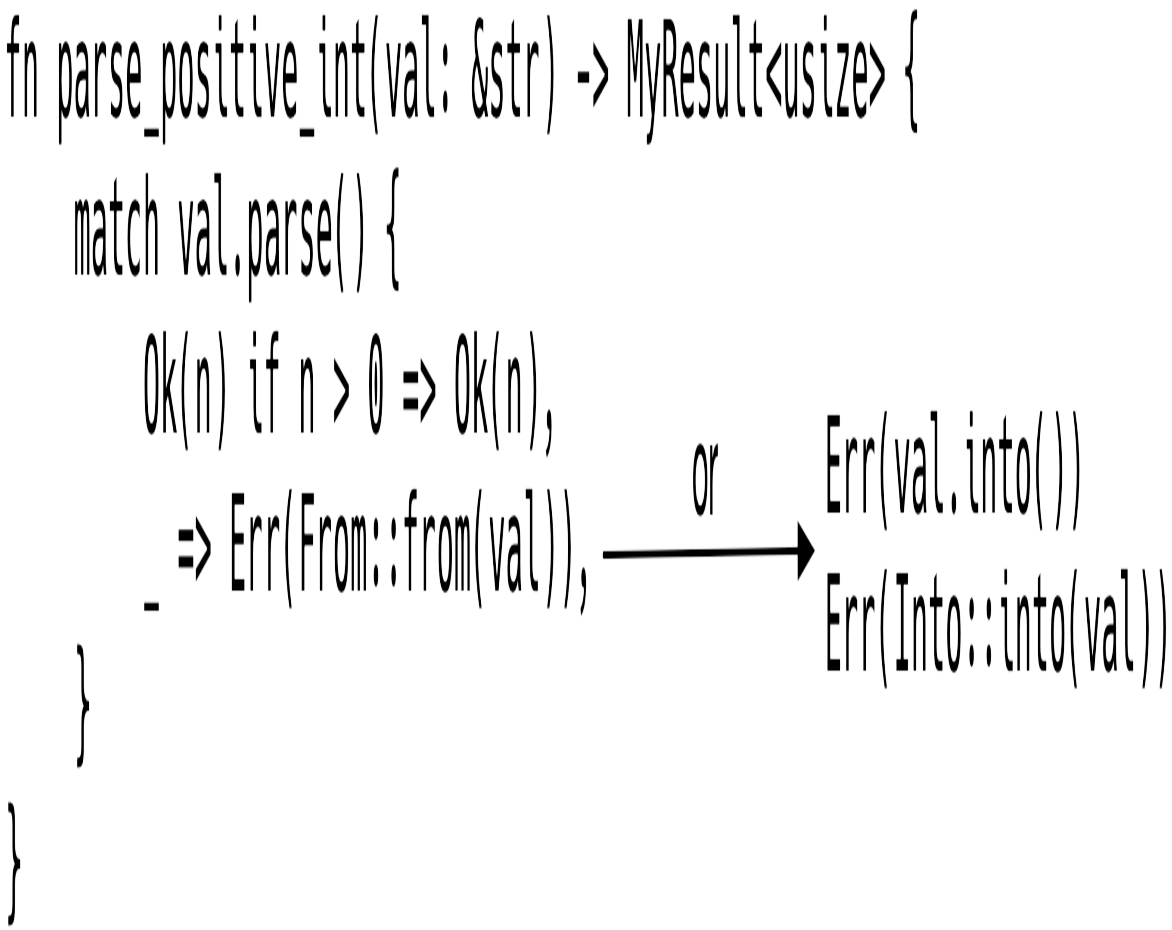


Figure 4-1. Alternate ways to convert a `&str` to an `Error` using `From` and `Into` traits

Now that you have a way to convert a string to a number, integrate it into your `get_args`. See if you can get your program to print a usage like the following. Note that I use the short and long names from the GNU version:

```
$ cargo run -- -h
headr 0.1.0
Ken Youens-Clark <kyclark@gmail.com>
Rust head

USAGE:
```

```
    headr [OPTIONS] <FILE>...

FLAGS:
    -h, --help       Prints help information
    -V, --version    Prints version information

OPTIONS:
    -c, --bytes <BYTES>    Number of bytes
    -n, --lines <LINES>    Number of lines [default: 10]

ARGS:
    <FILE>...    Input file(s) [default: -]
```

Run the program with no inputs and verify the defaults are correctly set:

```
$ cargo run
Config {
    files: [ ❶
        "-",
    ],
    lines: 10, ❷
    bytes: None, ❸
}
```

❶ The `files` should default to the filename "-".

❷ The number of `lines` should default to 10.

❸ The `bytes` should be `None`.

Run the program with arguments and ensure they are correctly parsed:

```
$ cargo run -- -n 3 tests/inputs/one.txt
Config {
    files: [
        "tests/inputs/one.txt", ❶
    ],
    lines: 3, ❷
    bytes: None, ❸
}
```

❶ The positional argument *tests/inputs/one.txt* is parsed as one of the `files`.

❷ The `-n` option for `lines` sets this to 3.

❸ The `-b` option for `bytes` defaults to `None`.

If I provide more than one positional argument, they will all go into the `files`, and the `-c` argument will go into `bytes`. In the following command, I'm again relying on the `bash` shell to expand the file glob **.txt* into all the files ending in *.txt*. PowerShell users should refer to the equivalent use of `Get-ChildItem` shown in Chapter 3:

```
$ cargo run -- -c 4 tests/inputs/*.txt
Config {
    files: [
        "tests/inputs/empty.txt", ❶
        "tests/inputs/one.txt",
        "tests/inputs/three.txt",
        "tests/inputs/two.txt",
    ],
    lines: 10, ❷
    bytes: Some( ❸
        4,
    ),
}
```

❶ There are four files ending in *.txt*.

❷ The `lines` is still set to the default value of `10`.

❸ The `-c 4` results in the `bytes` now being `Some(4)`.

Any value for `-n` or `-c` that cannot be parsed into a positive integer should cause the program to halt with an error:

```
$ cargo run -- -n blarg tests/inputs/one.txt
illegal line count -- blarg
$ cargo run -- -c 0 tests/inputs/one.txt
illegal byte count -- 0
```

The program should disallow `-n` and `-c` to be present together:

```
$ cargo run -- -n 1 -c 1 tests/inputs/one.txt
```

```
error: The argument '--lines <LINES>' cannot be used with '--bytes
<BYTES>'
```

Just parsing and validating the arguments is a challenge, but I know you can do it. Be sure to consult the `clap` documentation as you figure this out. I recommend you not move forward until your program can pass all the tests included with **`cargo test dies`**:

```
running 3 tests
test dies_bad_lines ... ok
test dies_bad_bytes ... ok
test dies_bytes_and_lines ... ok
```

Defining the Arguments

Following is how I defined the arguments for `clap`. Note that the two options for `lines` and `bytes` will take values. This is different from the flags implemented in Chapter 3 that are used as Boolean values:

```
let matches = App::new("headr")
    .version("0.1.0")
    .author("Ken Youens-Clark <kyclark@gmail.com>")
    .about("Rust head")
    .arg(
        Arg::with_name("lines") ❶
            .short("n")
            .long("lines")
            .value_name("LINES")
            .help("Number of lines")
            .default_value("10"),
    )
    .arg(
        Arg::with_name("bytes") ❷
            .short("c")
            .long("bytes")
            .value_name("BYTES")
            .takes_value(true)
            .conflicts_with("lines")
            .help("Number of bytes"),
    )
    .arg(
        Arg::with_name("files") ❸
            .value_name("FILE")
```

```
                .help("Input file(s)")
                .required(true)
                .default_value("-")
                .min_values(1),
        )
        .get_matches();
```

- ❶ The `lines` option takes a value and defaults to “10.”
- ❷ The `bytes` option takes a value, and it conflicts with the `lines` parameter so that they are mutually exclusive.
- ❸ The `files` parameter is positional, required, takes one or more values, and defaults to “-”.

> **NOTE**
>
> The `Arg::value_name` will be printed in the usage documentation, so be sure to choose a descriptive name. Don’t confuse this with the `Arg::with_name` that uniquely defines the name of the argument for accessing within your code.

Following is how I can use `parse_positive_int` inside `get_args` to validate `lines` and `bytes`. When the function returns an `Err` variant, I use `?` to propagate the error to `main` and end the program; otherwise, I return the `Config`:

```
pub fn get_args() -> MyResult<Config> {
    let matches = App::new("headr")... // Same as before

    let lines = matches
        .value_of("lines") ❶
        .map(parse_positive_int) ❷
        .transpose() ❸
        .map_err(|e| format!("illegal line count -- {}", e))?; ❹

    let bytes = matches ❺
        .value_of("bytes")
        .map(parse_positive_int)
        .transpose()
        .map_err(|e| format!("illegal byte count -- {}", e))?;
```

```
    Ok(Config {
        files: matches.values_of_lossy("files").unwrap(), ❻
        lines: lines.unwrap(), ❼
        bytes ❽
    })
}
```

❶ `ArgMatches.value_of` returns an `Option<&str>`.

❷ Use `Option::map` to unpack a `&str` from `Some` and send it to `parse_positive_int`.

❸ The result of `Option::map` will be an `<Option<Result>>`, and `Option::transpose` will turn this into a `<Result<Option>>`.

❹ In the event of an `Err`, create an informative error message. Use `?` to propagate an `Err` or unpack the `Ok` value.

❺ Do the same for `bytes`.

❻ The `files` option should have at least one value and so should be safe to call `Option::unwrap`.

❼ The `lines` has a default value and is safe to unwrap.

❽ The `bytes` should be left as an `Option`. Use the `struct` field init shorthand since the name of the field is the same as the variable.

In the preceding code, I could have written the `Config` with every key/value pair like so:

```
Ok(Config {
    files: matches.values_of_lossy("files").unwrap(),
    lines: lines.unwrap(),
    bytes: bytes,
})
```

Clippy will suggest the following:

```
$ cargo clippy
warning: redundant field names in struct initialization
  --> src/lib.rs:61:9
   |
```

```
61 |         bytes: bytes,
   |         ^^^^^^^^^^^^ help: replace it with: `bytes`
   |
   = note: `#[warn(clippy::redundant_field_names)]` on by default
   = help: for further information visit https://rust-
lang.github.io/
     rust-clippy/master/index.html#redundant_field_names
```

It’s quite a bit of work to validate all the user input, but now I have some assurance that I can proceed with good data.

Processing the Input Files

This challenge program should handle the input files just as in Chapter 3, so I suggest you bring in the `open` function from there:

```
fn open(filename: &str) -> MyResult<Box<dyn BufRead>> {
    match filename {
        "-" => Ok(Box::new(BufReader::new(io::stdin()))),
        _ => Ok(Box::new(BufReader::new(File::open(filename)?))),
    }
}
```

Be sure to add all the require dependencies:

```
use clap::{App, Arg};
use std::error::Error;
use std::fs::File;
use std::io::{self, BufRead, BufReader, Read};
```

Expand your `run` function to try opening the files, printing errors as you encounter them:

```
pub fn run(config: Config) -> MyResult<()> {
    for filename in config.files { ❶
        match open(&filename) { ❷
            Err(err) => eprintln!("{}: {}", filename, err), ❸
            Ok(_file) => println!("Opened {}", filename), ❹
        }
    }
    Ok(())
}
```

❶ Iterate through each of the filenames.

❷ Attempt to open the filename.

❸ Print errors to `STDERR`.

❹ Print a message that the file was successfully opened.

Run your program with a good file and a bad file to ensure it seems to work:

```
$ cargo run -- blargh tests/inputs/one.txt
blargh: No such file or directory (os error 2)
Opened tests/inputs/one.txt
```

Next, try to solve reading the lines and then bytes of a given file, then try to add the headers separating multiple file arguments. Look closely at the error output from `head` when handling invalid files. Notice that readable files have a header first and then the file output, but invalid files only print an error. Additionally, there is an extra blank line separating the output for the valid files:

```
$ head -n 1 tests/inputs/one.txt blargh tests/inputs/two.txt
==> tests/inputs/one.txt <==
Öne line, four words.
head: blargh: No such file or directory

==> tests/inputs/two.txt <==
Two lines.
```

I've specifically designed some challenging inputs for you to consider. To see what you face, use the `file` command to report file type information:

```
$ file tests/inputs/*.txt
tests/inputs/empty.txt: empty ❶
tests/inputs/one.txt:   UTF-8 Unicode text ❷
tests/inputs/three.txt: ASCII text, with CRLF, LF line terminators
❸
tests/inputs/two.txt:   ASCII text ❹
```

❶ This is an empty file just to ensure your program doesn't fall over.

❷ This file contains Unicode as I put an umlaut over the O in *Őne* to force you to consider the differences between bytes and characters.

❸ This file has Windows-style line endings.

❹ This file has Unix-style line endings.

> **TIP**
>
> On Windows, the newline is the combination of the carriage return and the line feed, often shown as CRLF or `\r\n`. On Unix platforms, only the newline is used, so LF or `\n`. These line endings must be preserved in the output from your program, so you will have to find a way to read the lines in a file without removing the line endings.

Reading Bytes versus Characters

I want to explain the difference between reading *bytes* and *characters* from a file. In the early 1960s, the American Standard Code for Information Interchange (ASCII, pronounced *as-key*) table of 128 characters represented all possible text elements in computing. It only takes seven bits ($2^7 = 128$) to represent each character, so the notion of byte and character were interchangeable.

Since the creation of Unicode (Universal Coded Character Set) to represent all the writing systems of the world (and even emojis), some characters may require up to four bytes. The Unicode standard defines several ways to encode characters including the UTF-8 (Unicode Transformation Format using 8 bits). As I noted, the file *tests/inputs/one.txt* begins with the character *Ő* which is two bytes long in UTF-8. If you want `head` to show you this one character, you must request two bytes:

```
$ head -c 2 tests/inputs/one.txt
Ö
```

If I ask `head` to select just the first byte from this file, I get the byte value `195`, which is not a valid UTF-8 string. The output is a special character that indicates a problem converting a character into Unicode:

```
$ head -c 1 tests/inputs/one.txt
�
```

The challenge program is expected to recreate this behavior. This is a challenging program to write, but you should be able to use `std::io`, `std::fs::File`, and `std::io::BufReader` to figure out how to read bytes and lines from each of the files. I've included a full set of tests in *tests/cli.rs* that you should have copied into your source tree. Be sure to run **`cargo test`** frequently to check your progress. Do your best to pass all the tests before looking at my solution.

Solution

I was really surprised by how much I learned by writing this program. What I expected to be a rather simple program proved to be very challenging. I'd like to step you through how I arrived at my solution, starting with how I read a file line-by-line.

Reading a File Line-by-line

To start, I will modify some code from Chapter 3 for reading the lines from a file:

```
pub fn run(config: Config) -> MyResult<()> {
    for filename in config.files {
        match open(&filename) {
            Err(err) => eprintln!("{}: {}", filename, err),
            Ok(file) => {
                for line in file.lines().take(config.lines) { ❶
                    println!("{}", line?); ❷
                }
            }
        }
    }
    Ok(())
}
```

❶ Take the desired number of lines from the filehandle.

❷ Print the line to the console.

I think this is a really fun solution because it uses the `Iterator::take` method to select the number of lines from `config.lines`. I can run the program to select one line from a file that contains three, and it appears to work grandly:

```
$ cargo run -- -n 1 tests/inputs/three.txt
Three
```

If I run **`cargo test`**, the program will pass several tests, which seems pretty good for having only implemented a small portion of the specs. It's failing all the tests starting with *three* which use the Windows-encoded input file. To fix this problem, I have a confession to make.

Preserving Line Endings While Reading a File

It pains me to tell you this, dear reader, but I lied to you in Chapter 3. The `catr` program I showed does not completely replicate the original program because it uses `BufRead::lines` to read the input files. The documentation for that functions says "Each string returned will **not** have a newline byte (the `0xA` byte) or CRLF (`0xD`, `0xA` bytes) at the end." I hope you'll forgive me because I wanted to show you how easy it can be to read the lines of a file, but you should be aware that the `catr` program replaces Windows CRLF line endings with Unix-style newlines.

To fix this, I must instead use `BufRead::read_line`, which says "This function will read bytes from the underlying stream until the newline delimiter (the `0xA` byte) or EOF[1] is found. Once found, all bytes up to, and including, the delimiter (if found) will be appended to `buf`." Following is a version that will preserve the original line endings. With these changes, the program will pass more tests than it fails:

```
pub fn run(config: Config) -> MyResult<()> {
    for filename in config.files {
        match File::open(&filename) {
```

```
                Err(err) => eprintln!("{}: {}", filename, err),
                Ok(mut file) => { ❶
                    let mut line = String::new(); ❷
                    for _ in 0..config.lines { ❸
                        let bytes = file.read_line(&mut line)?; ❹
                        if bytes == 0 { ❺
                            break;
                        }
                        print!("{}", line); ❻
                        line.clear(); ❼
                    }
                }
            };
        }
        Ok(())
    }
```

❶ Accept the filehandle as a `mut` (mutable) value.

❷ Use `String::new` to create a new, empty mutable string buffer to hold each line.

❸ Use `for` to iterate through a `std::ops::Range` to count up from 0 to the requested number of lines. The variable name _ indicates I do not intend to use it.

❹ Use `BufRead::read_line` to read the next line.

❺ The filehandle will return 0 bytes when it reaches the end, so `break` out of the loop.

❻ Print the line including the original line ending.

❼ Use `String::clear` to empty the line buffer.

If I run **`cargo test`** at this point, I'm passing almost all the tests for reading lines and failing all those for reading bytes and handling multiple files.

Reading Bytes from a File

Next, I'll handle reading bytes from a file. After I attempt to open the file, I check to see if the `config.bytes` is `Some` number of bytes; otherwise, I'll

use the preceding code that reads lines:

```
for filename in config.files {
    match File::open(&filename) {
        Err(err) => eprintln!("{}: {}", filename, err),
        Ok(mut file) => {
            if let Some(num_bytes) = config.bytes { ❶
                let mut handle = file.take(num_bytes as u64); ❷
                let mut buffer = vec![0; num_bytes]; ❸
                let n = handle.read(&mut buffer)?; ❹
                print!("{}",
String::from_utf8_lossy(&buffer[..n])); ❺
            } else {
                ... // Read lines as before
            }
        }
    };
}
```

❶ Use pattern matching to check if `config.bytes` is `Some` number of bytes to read.

❷ Use `take` to read the requested number of bytes.

❸ Create a mutable buffer of a fixed length `num_bytes` filled with zeros to hold the bytes read from the file.

❹ Read the desired number of bytes from the filehandle into the buffer. The value `n` will report the number of bytes that were actually read, which may be fewer than the number requested.

❺ Convert the bytes into a string that may not be valid UTF-8. Note the range operation to select only the bytes actually read.

> **TIP**
>
> The `take` method from the `std::io::Read` trait expects its argument to be the type `u64`, but I have a `usize`. I *cast* or convert the value using the `as` keyword.

This was perhaps the hardest part of the program for me. Once I figured out how to read only a few bytes, I had to figure out how to convert them to text.

If I take only part of a multibyte character, the result will fail because strings in Rust must be valid UTF-8. I was happy to find `String::from_utf8_lossy` that will quietly convert invalid UTF-8 sequences to the *unknown* or *replacement* character:

```
$ cargo run -- -c 1 tests/inputs/one.txt
�
```

Let me show you the first way I tried to read the bytes from a file. I decided to read the entire file into a string, convert that into a vector of bytes, and use a slice to select the first `num_bytes`.

```
let mut contents = String::new(); ❶
file.read_to_string(&mut contents)?; // Danger here ❷
let bytes = contents.as_bytes(); ❸
print!("{}", String::from_utf8_lossy(&bytes[..num_bytes])); // More
danger ❹
```

❶ Create a new string buffer to hold the contents of the file.

❷ Read the entire file contents into the string buffer.

❸ Use `str::as_bytes` to convert the contents into bytes (`u8` or unsigned 8-bit integers).

❹ Use `String::from_utf8_lossy` to turn a slice of the `bytes` into a string.

> **WARNING**
>
> I show you this approach so that you know how to read a file into a string; however, this can be a very dangerous thing to do if the file's size exceeds the amount of memory on your machine. In general, this is a terrible idea unless you are positive that a file is small.

Another serious problem with the preceding code is that it assumes the slice operation `bytes[..num_bytes]` will succeed. If you use this code with an empty file, for instance, you'll be asking for bytes that don't exist. This will cause your program to `panic` and exit immediately with an error

message:

```
$ cargo run -- -c 1 tests/inputs/empty.txt
thread 'main' panicked at 'range end index 1 out of range for slice
of
length 0', src/lib.rs:80:50
note: run with `RUST_BACKTRACE=1` environment variable to display a
backtrace
```

TIP

Rust can prevent you from making all sorts of egregious errors, but it can't stop you from doing stupid things. There are still plenty of ways for you to shoot yourself in the foot.

Following is perhaps the shortest way to read the desired number of bytes from a file:

```
let bytes: Result<Vec<_>, _> =
file.bytes().take(num_bytes).collect();
print!("{}", String::from_utf8_lossy(&bytes?));
```

In the preceding code, the type annotation `Result<Vec<_>, _>` is necessary as the compiler infers the type of `bytes` as a slice, which has an unknown size. I must indicate I want a `Vec`, which is a smart pointer to heap-allocated memory. The underscores (_) here indicate partial type annotation, which basically instructs the compiler to infer the types. Without this, the compiler complains thusly:

```
   Compiling headr v0.1.0 (/Users/kyclark/work/sysprog-
rust/playground/headr)
error[E0277]: the size for values of type `[u8]` cannot be known at
compilation time
  --> src/lib.rs:95:58
   |
95 |                     print!("{}",
String::from_utf8_lossy(&bytes?));
   |
^^^^^^^ doesn't
   |                                          have a size known at
```

```
compile-time
   |
   = help: the trait `Sized` is not implemented for `[u8]`
   = note: all local variables must have a statically known size
   = help: unsized locals are gated as an unstable feature
```

> **NOTE**
>
> You've now seen that the underscore _ serves various different functions. As the prefix or name of a variable, it shows the compiler you don't want to use the value. In a `match` arm, it is the wildcard for handling any case. When used in a type annotation, it tells the compiler to infer the type.

You can also indicate the type information on the righthand side of the expression using the *turbofish* operator (`::<>`). Often it's a matter of style whether you indicate the type on the lefthand or righthand side, but later you will see examples where the turbofish is required for some expressions:

```
let bytes = file.bytes().take(num_bytes).collect::<Result<Vec<_>,
_>>();
```

The unknown character produced by `String::from_utf8_lossy` (`b'\xef\xbf\xbd'`) is not exactly the same output produced by BSD `head` (`b'\xc3'`), making this somewhat difficult to test. If you look at the `run` helper function in *tests/cli.rs*, you'll see that I read the expected value (the output from `head`) and use the same function to convert what could be invalid UTF-8 so that I can compare the two outputs. The `run_stdin` function works similarly:

```
fn run(args: &[&str], expected_file: &str) -> TestResult {
    // Extra work here due to lossy UTF
    let mut file = File::open(expected_file)?;
    let mut buffer = Vec::new();
    file.read_to_end(&mut buffer)?;
    let expected = String::from_utf8_lossy(&buffer); ❶

    Command::cargo_bin(PRG)?
        .args(args)
        .assert()
```

```
            .success()
            .stdout(predicate::eq(&expected.as_bytes() as &[u8])); ❷

        Ok(())
    }
```

❶ Handle any invalid UTF-8 in the `expected_file`.

❷ Compare the output and expected values as a slice of bytes (`[u8]`).

Printing the File Separators

The last piece to handle is the separators between multiple files. As noted before, valid files have a header the puts the filename inside `==>` and `<==` markers. Files after the first have an additional newline at the beginning to visually separate the output. This means I will need to know the number of the file that I'm handling, which I can get by using the `Iterator::enumerate` method. Following is the final version of my `run` function that will pass all the tests:

```
pub fn run(config: Config) -> MyResult<()> {
    let num_files = config.files.len(); ❶

    for (file_num, filename) in config.files.iter().enumerate() {
❷
        match File::open(&filename) {
            Err(err) => eprintln!("{}: {}", filename, err),
            Ok(file) => {
                if num_files > 1 { ❸
                    println!(
                        "{}==> {} <==",
                        if file_num > 0 { "\n" } else { "" }, ❹
                        filename
                    );
                }

                if let Some(num_bytes) = config.bytes {
                    let mut handle = file.take(num_bytes as u64);
                    let mut buffer = vec![0; num_bytes];
                    let n = handle.read(&mut buffer)?;
                    print!("{}",
String::from_utf8_lossy(&buffer[..n]));
                } else {
```

```
                    let mut line = String::new();
                    for _ in 0..config.lines {
                        let bytes = file.read_line(&mut line)?;
                        if bytes == 0 {
                            break;
                        }
                        print!("{}", line);
                        line.clear();
                    }
                }
            }
        };
    }

    Ok(())
}
```

❶ Use the `Vec::len` method to get the number of files.

❷ Use the `Iterator::enumerate` method to track both the file number and filenames.

❸ Only print headers when there are multiple files.

❹ Print a newline when the `file_num` is greater than 0, which indicates the first file.

Going Further

- Implement the multiplier suffixes of the GNU version so that, for instance, `-c=1K` means print the first 1024 bytes of the file. Be sure to add and run tests.
- Implement the negative number options from the GNU version where `-n=-3` means you should print all but the last three lines of the file. As always, create tests to ensure your program is correct.
- Add an option for selecting characters.
- Add the file with the Windows line endings to the tests in Chapter 3. Edit the *mk-outs.sh* for that program to incorporate this file, and then expand the tests and program to ensure that line endings are

preserved.

Summary

This chapter dove into some fairly sticky subjects such as converting types like a `&str` to a `usize`, a `String` to an `Error`, and a `usize` to a `u64`. I feel like it took me quite a while to understand the differences between `&str` and `String` and why I need to use `From::from` to create the `Err` part of `MyResult`. If you still feel confused, just know that you won't always. I think if you keep reading the docs and writing more code, it will eventually come to you.

Here are some things you accomplished in this exercise:

- You learned to create optional parameters that can take values. Previously, the options were flags.
- You saw that all command-line arguments are strings. You used the `str::parse` method to attempt the conversion of a string like "3" into the number `3`.
- You learned how to write and run a unit test for an individual function.
- You learned to convert types using the `as` keyword or with traits like `From` and `Into`.
- You found that _ as the name or prefix of a value is a way to indicate to the compiler that you don't intend to use a variable. When used in a type annotation, it tells the compiler to infer the type.
- You learned to that a `match` arm can incorporate an additional Boolean condition called a *guard*.
- You learned how to use `BufRead::read_line` to preserve line endings while reading a filehandle.

- You found that the `take` method works on both iterators and filehandles to limit the number of elements you select.
- You learned to indicate type information on the lefthand side of an assignment or on the righthand side using the turbofish.

---

1 EOF is an acronym for *end of file*.

Chapter 5. Word To Your Mother

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 5th Chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the author at kyclark@gmail.com.

You could count on me with just one hand

—They Might Be Giants

The venerable `wc` (*word count*) program dates back to version 1 of AT&T UNIX. This program will display the number of lines, words, and bytes found in some text from `STDIN` or one or more files.

In this exercise, you will learn:

- How to use the `Iterator::all` function
- How to create a module for tests
- How to fake a filehandle for testing
- How to conditionally format and print a value
- How to break a line of text into words and characters
- How to use `Iterator::collect` to turn an iterator into a vector

How wc Works

Here is an excerpt from the BSD `wc` manual page that describes how the tool works:

```
WC(1)                     BSD General Commands Manual
WC(1)

NAME
     wc -- word, line, character, and byte count

SYNOPSIS
     wc [-clmw] [file ...]

DESCRIPTION
     The wc utility displays the number of lines, words, and bytes
contained
     in each input file, or standard input (if no file is
specified) to the
     standard output.  A line is defined as a string of characters
delimited
     by a <newline> character.  Characters beyond the final
<newline> charac-
     ter will not be included in the line count.

     A word is defined as a string of characters delimited by white
space
     characters.  White space characters are the set of characters
for which
     the iswspace(3) function returns true.  If more than one input
file is
     specified, a line of cumulative counts for all the files is
displayed on
     a separate line after the output for the last file.

     The following options are available:

     -c      The number of bytes in each input file is written to
the standard
             output.  This will cancel out any prior usage of the -
m option.

     -l      The number of lines in each input file is written to
the standard
             output.
```

```
     -m      The number of characters in each input file is written
to the
             standard output.  If the current locale does not
support multi-
             byte characters, this is equivalent to the -c option.
This will
             cancel out any prior usage of the -c option.

     -w      The number of words in each input file is written to
the standard
             output.

     When an option is specified, wc only reports the information
requested by
     that option.  The order of output always takes the form of
line, word,
     byte, and file name.  The default action is equivalent to
specifying the
     -c, -l and -w options.

     If no files are specified, the standard input is used and no
file name is
     displayed.  The prompt will accept input until receiving EOF,
or [^D] in
     most environments.
```

A picture is worth a kilobyte of words, so I'll show you some examples using the test files in the *05_wcr* directory. First, consider an empty file that should report 0 lines, words, and bytes, each of which should be right-justified in a field 8 characters wide:

```
$ cd 05_wcr
$ wc tests/inputs/empty.txt
       0       0       0 tests/inputs/empty.txt
```

Next, try a file with one line of text. Note that I've put varying amounts of spaces in between some words and a tab character for reasons that will be discussed later. I will use the `cat` with the flags `-t` to display the tab character as `^I` and the `-e` to display `$` for the end of the line:

```
$ cat -te tests/inputs/fox.txt
The  quick brown fox^Ijumps over   the lazy dog.$
```

This example is short enough that I can manually count all the lines, words, bytes as shown in Figure 5-1 where spaces are noted with raised dots, the tab character with `\t`, and the end-of-line as `$`.

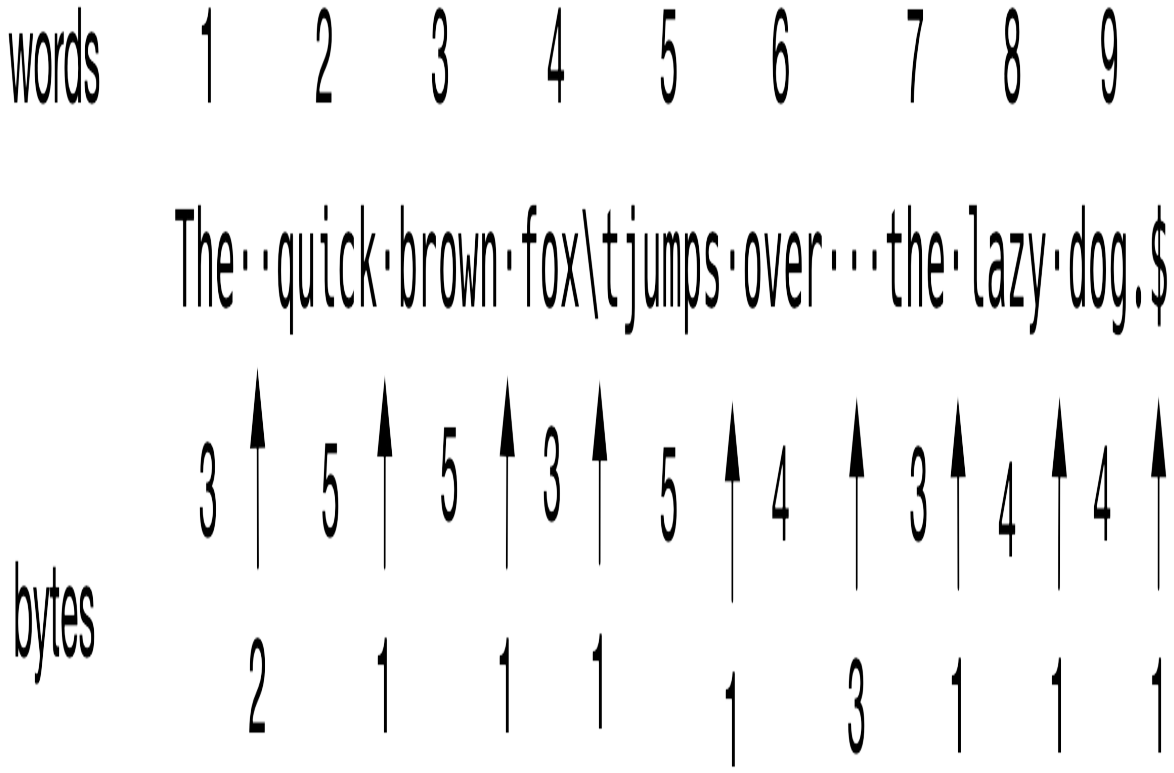


Figure 5-1. There is one line of text containing nine words comprised of 48 bytes

I find that `wc` is in agreement:

```
$ wc tests/inputs/fox.txt
       1       9      48 tests/inputs/fox.txt
```

As mentioned in Chapter 3, *bytes* may equate to *characters* for ASCII, but Unicode characters can take multiple bytes. The file *tests/inputs/atlamal.txt* contains the first stanza from *Atlamál hin groenlenzku* or *The Greenland Ballad of Atli*, an Old Norse poem[1]:

```
$ cat tests/inputs/atlamal.txt
Frétt hefir öld óvu, þá er endr of gerðu
```

```
seggir samkundu, sú var nýt fæstum,
æxtu einmæli, yggr var þeim síðan
ok it sama sonum Gjúka, er váru sannráðnir.
```

According to `wc`, this file contains 4 lines, 29 words, and 177 bytes:

```
$ wc tests/inputs/atlamal.txt
       4      29     177 tests/inputs/atlamal.txt
```

If I only wanted the number of *lines*, I can use the `-l` flag and only that column will be shown:

```
$ wc -l tests/inputs/atlamal.txt
       4 tests/inputs/atlamal.txt
```

I can similarly request only the number of *bytes* with `-c` or *words* with `-w`, and only those two columns will be shown:

```
$ wc -w -c tests/inputs/atlamal.txt
      29     177 tests/inputs/atlamal.txt
```

I can request the number of *characters* using the `-m` flag:

```
$ wc -m tests/inputs/atlamal.txt
     159 tests/inputs/atlamal.txt
```

The GNU version of `wc` will show both character and byte counts if you provide both the flags `-m` and `-c`, but the BSD version will only show one or the other with the latter flag taking precedence:

```
$ wc -cm tests/inputs/atlamal.txt ❶
     159 tests/inputs/atlamal.txt
$ wc -mc tests/inputs/atlamal.txt ❷
     177 tests/inputs/atlamal.txt
```

❶ The `-m` flag comes last, so characters are shown.

❷ The `-c` flag comes last, so bytes are shown.

Note that no matter the order of the flags like `-wc` or `-cw`, the output columns are always ordered by lines, words, and bytes/characters:

```
$ wc -cw tests/inputs/atlamal.txt
      29     177 tests/inputs/atlamal.txt
```

If no positional arguments are provided, `wc` will read from `STDIN` and will not print a filename:

```
$ cat tests/inputs/atlamal.txt | wc -lc
       4     173
```

The GNU version of `wc` will understand the filename `-` to mean `STDIN`, and it also provides long flag names as well as some other options:

```
$ wc --help
Usage: wc [OPTION]... [FILE]...
  or:  wc [OPTION]... --files0-from=F
Print newline, word, and byte counts for each FILE, and a total
line if
more than one FILE is specified.  With no FILE, or when FILE is -,
read standard input.  A word is a non-zero-length sequence of
characters
delimited by white space.
The options below may be used to select which counts are printed,
always in
the following order: newline, word, character, byte, maximum line
length.
  -c, --bytes            print the byte counts
  -m, --chars            print the character counts
  -l, --lines            print the newline counts
      --files0-from=F    read input from the files specified by
                           NUL-terminated names in file F;
                           If F is - then read names from standard
input
  -L, --max-line-length  print the length of the longest line
  -w, --words            print the word counts
      --help     display this help and exit
      --version  output version information and exit
```

If processing more than one file, both versions will finish with a *total* line showing the number of lines, words, and bytes for all the inputs:

```
$ wc tests/inputs/*.txt
       4      29     177 tests/inputs/atlamal.txt
       0       0       0 tests/inputs/empty.txt
       1       9      48 tests/inputs/fox.txt
       5      38     225 total
```

Nonexistent files are noted with a warning to STDERR as the files are being processed:

```
$ wc tests/inputs/fox.txt blargh tests/inputs/atlamal.txt
       1       9      48 tests/inputs/fox.txt
wc: blargh: open: No such file or directory
       4      29     177 tests/inputs/atlamal.txt
       5      38     225 total
```

I can also redirect filehandle 2 in `bash` to verify that `wc` prints the warning to STDERR:

```
$ wc tests/inputs/fox.txt blargh tests/inputs/atlamal.txt 2>err ❶
       1       9      48 tests/inputs/fox.txt
       4      29     177 tests/inputs/atlamal.txt
       5      38     225 total
$ cat err ❷
wc: blargh: open: No such file or directory
```

❶ Redirect output handle 2 (STDERR) to the file *err*.

❷ Verify that the error message is in the file.

The preceding behavior is as much as the challenge program will be expected to implement. There is an extensive test suite to help you along the way.

Getting Started

The challenge program should be called `wcr` (pronounced *wick-er*) for our Rust version of `wc`. Use **`cargo new wcr`** to start, then modify your *Cargo.toml* to add the following dependencies:

```
[dependencies]
```

```
clap = "2.33"

[dev-dependencies]
assert_cmd = "1"
predicates = "1"
rand = "0.8"
```

Copy my *05_wcr/tests* directory into your new project and run **`cargo test`** to perform an initial build and run the tests, all of which should fail. I can be a rather unimaginative sort sometimes, so I'm going to keep using the same structure for *src/main.rs* that I've used in the previous programs:

```
fn main() {
    if let Err(e) = wcr::get_args().and_then(wcr::run) {
        eprintln!("{}", e);
        std::process::exit(1);
    }
}
```

Following is a skeleton for *src/lib.rs* you can copy. First, here is how I would define the `Config` to represent the command-line parameters:

```
use clap::{App, Arg};
use std::error::Error;

type MyResult<T> = Result<T, Box<dyn Error>>;

#[derive(Debug)]
pub struct Config {
    files: Vec<String>, ❶
    lines: bool, ❷
    words: bool, ❸
    bytes: bool, ❹
    chars: bool, ❺
}
```

❶ The `files` will be a vector of strings.

❷ The `lines` is a Boolean for whether to print the line count.

❸ The `words` is a Boolean for whether to print the word count.

❹ The `bytes` is a Boolean for whether to print the byte count.

❺ The `chars` is a Boolean for whether to print the character count.

Here are the two functions you'll need to get started. I'll let you fill in the `get_args` from this skeleton:

```
pub fn get_args() -> MyResult<Config> {
    let matches = App::new("wcr")
        .version("0.1.0")
        .author("Ken Youens-Clark <kyclark@gmail.com>")
        .about("Rust wc")
        // What goes here?
        .get_matches()

    Ok(Config {
        files: ...
        lines: ...
        words: ...
        bytes: ...
        chars: ...
    })
}
```

I suggest you start your `run` by printing the configuration:

```
pub fn run(config: Config) -> MyResult<()> {
    println!("{:#?}", config);
    Ok(())
}
```

Try to get your program to generate `--help` output like the following:

```
$ cargo run -- --help
wcr 0.1.0
Ken Youens-Clark <kyclark@gmail.com>
Rust wc

USAGE:
    wcr [FLAGS] [FILE]...

FLAGS:
    -c, --bytes      Show byte count
    -m, --chars      Show character count
    -h, --help       Prints help information
```

```
    -l, --lines      Show line count
    -V, --version    Prints version information
    -w, --words      Show word count

ARGS:
    <FILE>...    Input file(s) [default: -]
```

I fretted a bit about whether to mimic the BSD or GNU version of `wc` for combining the `-m` (character) and `-c` (bytes) flags. I decided to use the BSD behavior, so your program should disallow both of these flags used together:

```
$ cargo run -- -cm tests/inputs/fox.txt
error: The argument '--bytes' cannot be used with '--chars'

USAGE:
    wcr --bytes --chars
```

The default behavior will be to print lines, words, and bytes, which means those values in the configuration should be `true` when none have been explicitly requested by the user. Ensure your program will print this:

```
$ cargo run -- tests/inputs/fox.txt
Config {
    files: [ ❶
        "tests/inputs/fox.txt",
    ],
    lines: true,
    words: true,
    bytes: true,
    chars: false, ❷
}
```

❶ A positional argument should be found in the `files`.

❷ The `chars` value should be `false` unless the `-m|--chars` flag is present.

If any single flag is present, then all the other flags *not* mentioned should be `false`:

```
$ cargo run -- -l tests/inputs/*.txt ❶
```

```
Config {
    files: [
        "tests/inputs/atlamal.txt",
        "tests/inputs/empty.txt",
        "tests/inputs/fox.txt",
    ],
    lines: true, ❷
    words: false,
    bytes: false,
    chars: false,
}
```

❶ The `-l` flag indicates only the line count is wanted.

❷ The `lines` value is the only one that is `true`.

Stop here and get this much working. My dog needs a bath, so I'll be right back.

I guess you got that figured out, so following is the first part of my `get_args`. There's nothing new to how I declare the parameters, so I'll not comment on this:

```
pub fn get_args() -> MyResult<Config> {
    let matches = App::new("wcr")
        .version("0.1.0")
        .author("Ken Youens-Clark <kyclark@gmail.com>")
        .about("Rust wc")
        .arg(
            Arg::with_name("files")
                .value_name("FILE")
                .help("Input file(s)")
                .default_value("-")
                .min_values(1),
        )
        .arg(
            Arg::with_name("lines")
                .value_name("LINES")
                .help("Show line count")
                .takes_value(false)
                .short("l")
                .long("lines"),
        )
        .arg(
```

```
            Arg::with_name("words")
                .value_name("WORDS")
                .help("Show word count")
                .takes_value(false)
                .short("w")
                .long("words"),
        )
        .arg(
            Arg::with_name("bytes")
                .value_name("BYTES")
                .help("Show byte count")
                .takes_value(false)
                .short("c")
                .long("bytes"),
        )
        .arg(
            Arg::with_name("chars")
                .value_name("CHARS")
                .help("Show character count")
                .takes_value(false)
                .short("m")
                .long("chars")
                .conflicts_with("bytes"),
        )
        .get_matches();
```

After `clap` parses the arguments, I unpack them and try to figure out the default values:

```
    let mut lines = matches.is_present("lines"); ❶
    let mut words = matches.is_present("words");
    let mut bytes = matches.is_present("bytes");
    let mut chars = matches.is_present("chars");

    if [lines, words, bytes, chars].iter().all(|v| v == &false) {
❷
        lines = true;
        words = true;
        bytes = true;
        chars = false;
    }

    Ok(Config { ❸
        files: matches.values_of_lossy("files").unwrap(),
        lines,
        words,
```

```
            bytes,
            chars,
        })
    }
```

1. Unpack all the flags.
2. If all the flags are `false`, then set `lines`, `words`, and `bytes` to `true`.
3. Use the struct field init shorthand to set the values.

I want to highlight that I create a slice with all the flags and call the `slice::iter` method to create an iterator. This is so I can use the `Iterator::all` function to find if all the values are `false`. This method expects a closure, which is an anonymous function that can be passed as an argument to another function. Here, the closure is a *predicate* or a *test* that figures out if an element is `false`. I want to know if all the flags are `false`, and so a reference to each flag is passed as the argument to the closure. The values are references, so I must compare each value to `&false` which is a reference to a Boolean value. If *all* the evaluations are `true`, then `Iterator::all` will return `true` [2].

ITERATOR METHODS THAT TAKE A CLOSURE

You should take some time to read the `Iterator` documentation to note the other methods that take a closure as an argument to select, test, or transform the elements, including:

- `Iterator::any` will return `true` if even one evaluation of the closure for an item returns `true`.
- `Iterator::filter` will find all elements for which the predicate is `true`.
- `Iterator::map` will apply a closure to each element and return a `std::iter::Map` with the transformed elements.

- `Iterator::find` will return the first element of an iterator that satisfies the predicate as `Some(value)` or `None` if all elements evaluate to `false`.
- `Iterator::position` will return the index of the first element that satisfies the predicate as `Some(value)` or `None` if all elements evaluate to `false`.
- `Iterator::cmp`, `Iterator::min_by` and `Iterator::max_by` have predicates that accept pairs of items for comparison to find the minimum and maximum, respectively.

Iterating the Files

Now to work on the counting. You might start by processing each file, counting the various bits, and printing the desired columns. I suggest you once again use the `open` function from Chapter 2 for opening the files:

```
fn open(filename: &str) -> MyResult<Box<dyn BufRead>> {
    match filename {
        "-" => Ok(Box::new(BufReader::new(io::stdin()))),
        _ => Ok(Box::new(BufReader::new(File::open(filename)?))),
    }
}
```

Be sure to expand your imports to the following:

```
use clap::{App, Arg};
use std::error::Error;
use std::fs::File;
use std::io::{self, BufRead, BufReader};
```

Here is a `run` function to get you going:

```
pub fn run(config: Config) -> MyResult<()> {
    for filename in &config.files {
```

```
            match open(filename) {
                Err(err) => eprintln!("{}: {}", filename, err),
                Ok(_file) => println!("Opened {}", filename),
            }
        }

        Ok(())
    }
```

Using your open filehandle, start as simply as possible using the empty file and make sure your program prints zeros for the three columns of lines, words, and bytes:

```
$ cargo run -- tests/inputs/empty.txt
       0       0       0 tests/inputs/empty.txt
```

Next use *tests/inputs/fox.txt* and make sure you get the following counts. I specifically added various kinds and numbers of whitespace to challenge you on how to split the text into words:

```
$ cargo run -- tests/inputs/fox.txt
       1       9      48 tests/inputs/fox.txt
```

Be sure your program can handle the Unicode in *tests/inputs/atlamal.txt* correctly:

```
$ cargo run -- tests/inputs/atlamal.txt
       4      29     177 tests/inputs/atlamal.txt
```

And that you correctly count the characters:

```
$ cargo run -- tests/inputs/atlamal.txt -wml
       4      29     159 tests/inputs/atlamal.txt
```

When you can handle any one file, use more than one to check that you print the correct *total* column:

```
$ cargo run -- tests/inputs/*.txt
       4      29     177 tests/inputs/atlamal.txt
```

```
       0       0       0 tests/inputs/empty.txt
       1       9      48 tests/inputs/fox.txt
       5      38     225 total
```

When all that works correctly, try reading from `STDIN`:

```
$ cat tests/inputs/atlamal.txt | cargo run
       4      29     177
```

Run **`cargo test`** often to see how you're progressing. Don't read ahead until you pass all the tests.

Solution

I'd like to walk you through how I arrived at a solution, and I'd like to stress that mine is just one of many possible ways to write this program. As long as your code passes the tests and produces the same output as the BSD version of `wc`, then it works well and you should be proud of your accomplishments.

Counting the Elements of a File or STDIN

I suggested you start by iterating the filenames and printing the counts for the file. To that end, I decided to create a function called `count` that would take a filehandle and possibly return a struct called `FileInfo` containing the number of lines, words, bytes, and characters each represented as a `usize`. I say that the function will *possibly* return this struct because the function will involve IO, which could go sideways. I place the following definition just after the `Config` struct. For reasons I will explain shortly, this must derive the `PartialEq` trait in addition to `Debug`:

```
#[derive(Debug, PartialEq)]
pub struct FileInfo {
    num_lines: usize,
    num_words: usize,
    num_bytes: usize,
    num_chars: usize,
}
```

To represent this the function might succeed or fail, it will return a `MyResult<FileInfo>` meaning that on success it will have `Ok<FileInfo>` and on failure it will have an `Err`. To start this function, I will initialize some mutable variables to count all the elements and will return a `FileInfo` struct:

```
pub fn count(mut file: impl BufRead) -> MyResult<FileInfo> { ❶
    let mut num_lines = 0; ❷
    let mut num_words = 0;
    let mut num_bytes = 0;
    let mut num_chars = 0;

    Ok(FileInfo {
        num_lines, ❸
        num_words,
        num_bytes,
        num_chars,
    })
}
```

❶ The `count` function will accept a mutable `file` value, and it might return a `FileInfo` struct.

❷ Initialize mutable variables to count the lines, words, bytes, and characters.

❸ For now, return a `FileInfo` with all zeros.

> **NOTE**
>
> I'm introducing the `impl` keyword to indicate that the `file` value must implement the `BufRead` trait. Recall that `open` returns a value that meets this criteria. You'll shortly see how this makes the function flexible.

In Chapter 3, I showed how to write a unit test, placing it just after the function it was testing. I'm going to create a unit test for the `count` function, but this time I'm going to place it inside a module called `tests`. This is a tidy way to group unit tests, and I can use a configuration option that tells Rust to only compile the module during testing. This is especially useful

because I want to use `std::io::Cursor` in my test to make a fake filehandle for the `count` function. The module will not be included when I build and run the program when not testing.

A `Cursor` is "used with in-memory buffers, anything implementing `AsRef<[u8]>`, to allow them to implement `Read` and/or `Write`, allowing these buffers to be used anywhere you might use a reader or writer that does actual I/O." Following is how I create the `tests` module and then import and test the `count` function:

```
#[cfg(test)] ❶
mod tests { ❷
    use super::{count, FileInfo}; ❸
    use std::io::Cursor; ❹

    #[test]
    fn test_count() {
        let text = "I don't want the world. I just want your
half.\r\n";
        let info = count(Cursor::new(text)); ❺
        assert!(info.is_ok()); ❻
        let expected = FileInfo {
            num_lines: 1,
            num_words: 10,
            num_chars: 48,
            num_bytes: 48,
        };
        assert_eq!(info.unwrap(), expected); ❼
    }
}
```

❶ The `cfg` enables conditional compilation, so this module will only be compiled when testing.

❷ Define a new module (`mod`) called *tests* to contain test code.

❸ Import the `count` function and `FileInfo` struct from the parent module `super`, meaning *next above or higher* and refers to the module above `tests` that contains it.

❹ Import `std::io::Cursor`.

❺ Run `count` with the `Cursor`, which implements `BufRead`.

❻ Ensure the result is `Ok`.

❼ Compare the result to the expected value. This comparison requires `FileInfo` to derive `PartialEq`.

Run this test using **`cargo test test_count`**. You will see lots of warnings from the Rust compiler about unused variables or variables that do not need to be mutable. The most important result is that the test fails:

```
failures:

---- tests::test_count stdout ----
thread 'tests::test_count' panicked at 'assertion failed: `(left ==
right)`
  left: `FileInfo { num_lines: 0, num_words: 0, num_bytes: 0,
num_chars: 0 }`,
 right: `FileInfo { num_lines: 1, num_words: 10, num_bytes: 48,
 num_chars: 48 }`', src/lib.rs:146:9
```

Take some time to write the rest of the `count` function that will pass this test. I will take my squeaky clean dog for a walk and maybe have some tea.

OK, we're back. Now I'll show you how I wrote my `count` function. I know from Chapter 3 that `BufRead::lines` will remove the line endings, and I don't want that because newlines in Windows files are two bytes (`\r\n`) but Unix newlines are just one byte (`\n`). I can copy some code from Chapter 3 that uses `BufRead::read_line` instead to read each line into a buffer. Conveniently, this function tells me how many bytes have been read from the file:

```
pub fn count(mut file: impl BufRead) -> MyResult<FileInfo> {
    let mut num_lines = 0;
    let mut num_words = 0;
    let mut num_bytes = 0;
    let mut num_chars = 0;
    let mut line = String::new(); ❶

    loop { ❷
        let line_bytes = file.read_line(&mut line)?; ❸
        if line_bytes == 0 { ❹
```

```
                break;
            }
            num_bytes += line_bytes; ❺
            num_lines += 1; ❻
            num_words += line.split_whitespace().count(); ❼
            num_chars += line.chars().count(); ❽
            line.clear(); ❾
        }

        Ok(FileInfo {
            num_lines,
            num_words,
            num_bytes,
            num_chars,
        })
    }
```

❶ Create a mutable buffer to hold each `line` of text.

❷ Create an infinite `loop` for reading the filehandle.

❸ Try to read a line from the filehandle.

❹ End of file (EOF) has been reached when zero bytes are read, so `break` out of the loop.

❺ Add the number of bytes from this line to the `num_bytes` variable.

❻ Each time through the loop is a line, so increment `num_lines`.

❼ Use the `str::split_whitespace` method to break the string on whitespace and use `Iterator::count` to find the number of words.

❽ Use the `str::chars` method to break the string into Unicode characters and use `Iterator::count` to count the characters.

❾ Clear the `line` buffer for the next line of text.

NOTE

Earlier I stressed how this input file had a varying number and type of whitespace separating words. I did this in case you chose to iterate over the characters to find word boundaries. That is, you might have not found `str::split_whitespace` and instead used an iterator over `str::chars`. If you increment the number of words every time you find whitespace assuming there is only one between words, you might end up overcounting words. Instead, you would need to find runs of whitespace as the delimiter between words. Similarly, you might have been tempted to use a regular

expression which would also need to consider one or more whitespace characters.

With these changes, `test_count` passes. I'd like to see how this looks, so I can change `run` to print the results of successfully counting the elements of the file or print a warning to `STDERR` when the file can't be opened:

```
pub fn run(config: Config) -> MyResult<()> {
    for filename in &config.files {
        match open(filename) {
            Err(err) => eprintln!("{}: {}", filename, err),
            Ok(file) => {
                if let Ok(info) = count(file) { ❶
                    println!("{:?}", info); ❷
                }
            }
        }
    }

    Ok(())
}
```

❶ Attempt to get the counts from a file.

❷ Print the counts.

When I run it on one of the test inputs, it appears to work for a valid file:

```
$ cargo run -- tests/inputs/fox.txt
FileInfo { num_lines: 1, num_words: 9, num_bytes: 48, num_chars: 48
}
```

It even handles reading from `STDIN`:

```
$ cat tests/inputs/fox.txt | cargo run
FileInfo { num_lines: 1, num_words: 9, num_bytes: 48, num_chars: 48
}
```

Now to make the output meet the specifications.

Formatting the Output

To create the expected output, I can start by changing `run` to always print the lines, words, and bytes followed by the filename:

```
pub fn run(config: Config) -> MyResult<()> {
    for filename in &config.files {
        match open(filename) {
            Err(err) => eprintln!("{}: {}", filename, err),
            Ok(file) => {
                if let Ok(info) = count(file) {
                    println!(
                        "{:>8}{:>8}{:>8} {}", ❶
                        info.num_lines,
                        info.num_words,
                        info.num_bytes,
                        filename
                    );
                }
            }
        }
    }

    Ok(())
}
```

❶ Format the number of lines, words, and bytes into a right-justified field 8 characters wide.

If I run it with one input file, it's already looking pretty sweet:

```
$ cargo run -- tests/inputs/fox.txt
       1       9      48 tests/inputs/fox.txt
```

If I run **`cargo test fox`**, I pass one out of eight tests. Huzzah!

```
running 8 tests
test fox ... ok
test fox_bytes ... FAILED
test fox_chars ... FAILED
test fox_bytes_lines ... FAILED
test fox_words_bytes ... FAILED
test fox_words ... FAILED
```

```
test fox_words_lines ... FAILED
test fox_lines ... FAILED
```

Inspect *tests/cli.rs* to see what the passing test looks like. Note that the tests reference constant values declared at the top of the module:

```
const PRG: &str = "wcr";
const EMPTY: &str = "tests/inputs/empty.txt";
const FOX: &str = "tests/inputs/fox.txt";
const ATLAMAL: &str = "tests/inputs/atlamal.txt";
```

Again I have a `run` helper function to run my tests:

```
fn run(args: &[&str], expected_file: &str) -> TestResult {
    let expected = fs::read_to_string(expected_file)?; ❶
    Command::cargo_bin(PRG)? ❷
        .args(args)
        .assert()
        .success()
        .stdout(expected);
    Ok(())
}
```

❶ Try to read the `expected` output for this command.

❷ Run the `wcr` program with the given arguments. Assert that the program succeeds and that `STDOUT` matches the `expected` value.

The `fox` test is running `wcr` with the `FOX` input file and no options, comparing it to the contents of the expected output file which was generated using *05_wcr/mk-outs.sh*:

```
#[test]
fn fox() -> TestResult {
    run(&[FOX], "tests/expected/fox.txt.out")
}
```

Look at the next function in the file to see a failing test:

```
#[test]
```

```
fn fox_bytes() -> TestResult {
    run(&["--bytes", FOX], "tests/expected/fox.txt.c.out") ❶
}
```

❶ Run the `wcr` program with the same input file and the `--bytes` option.

When run with `--bytes`, my program should only print that column of output, but it always prints lines, words, and bytes. I decided to write a function called `format_field` in *src/lib.rs* that would conditionally return a formatted string or the empty string depending on a Boolean value:

```
fn format_field(value: usize, show: bool) -> String { ❶
    if show { ❷
        format!("{:>8}", value) ❸
    } else {
        "".to_string() ❹
    }
}
```

❶ The function accepts a `usize` value and a Boolean and returns a `String`.

❷ Check if the `show` value is `true`.

❸ Return a new string by formatting the number into a string 8 characters wide.

❹ Otherwise, return the empty string.

> **NOTE**
>
> Why does this function return a `String` and not a `str`? They're both *strings*, but a `str` is an immutable, fixed-length string. The string that will be returned from the function is dynamically generated at runtime, so I must use `String`, which is a growable, heap-allocated structure. Props to dynamic languages like Perl and Python that hide so many complexities of strings which turn out to be way more complicated than I ever considered.

I can expand my `tests` module to add a unit test for this:

```
#[cfg(test)]
mod tests {
    use super::{count, format_field, FileInfo}; ❶
    use std::io::Cursor;

    #[test]
    fn test_count() {} // Same as before

    #[test]
    fn test_format_field() {
        assert_eq!(format_field(1, false), ""); ❷
        assert_eq!(format_field(3, true), "       3"); ❸
        assert_eq!(format_field(10, true), "      10"); ❹
    }
}
```

❶ Add `format_field` to the imports.

❷ The function should return the empty string when `show` is `false`.

❸ Check width for a single-digit number.

❹ Check width for a double-digit number.

Here is how I use it in context where I also handle printing the empty string when reading from `STDIN`:

```
pub fn run(config: Config) -> MyResult<()> {
    for filename in &config.files {
        match open(filename) {
            Err(err) => eprintln!("{}: {}", filename, err),
            Ok(file) => {
                if let Ok(info) = count(file) {
                    println!(
                        "{}{}{}{}{}", ❶
                        format_field(info.num_lines, config.lines),
                        format_field(info.num_words, config.words),
                        format_field(info.num_bytes, config.bytes),
                        format_field(info.num_chars, config.chars),
                        if filename.as_str() == "-" { ❷
                            "".to_string()
                        } else {
                            format!(" {}", filename)
                        }
                    );
                }
```

```
                }
            }
        }

        Ok(())
    }
```

❶ Format the output for each of the columns using the `format_field` function.

❷ When the filename is "-", print the empty string; otherwise, print a space and the filename.

With these changes, all the tests for **`cargo test fox`** pass. If I run the entire test suite, I'm still failing the *all* tests:

```
failures:
    test_all
    test_all_bytes
    test_all_bytes_lines
    test_all_lines
    test_all_words
    test_all_words_bytes
    test_all_words_lines
```

Look at the `all` function in *tests/cli.rs* to see that the test is using all the input files as arguments:

```
#[test]
fn all() -> TestResult {
    run(&[EMPTY, FOX, ATLAMAL], "tests/expected/all.out")
}
```

If I run my current program with all the input files, I can see that I'm missing the *total* line:

```
$ cargo run -- tests/inputs/*.txt
       4      29     177 tests/inputs/atlamal.txt
       0       0       0 tests/inputs/empty.txt
       1       9      48 tests/inputs/fox.txt
```

Here is my final `run` function that keeps a running total and prints those values when there is more than one input:

```
pub fn run(config: Config) -> MyResult<()> {
    let mut total_lines = 0; ❶
    let mut total_words = 0;
    let mut total_bytes = 0;
    let mut total_chars = 0;

    for filename in &config.files {
        match open(filename) {
            Err(err) => eprintln!("{}: {}", filename, err),
            Ok(file) => {
                if let Ok(info) = count(file) {
                    println!(
                        "{}{}{}{}{}",
                        format_field(info.num_lines, config.lines),
                        format_field(info.num_words, config.words),
                        format_field(info.num_bytes, config.bytes),
                        format_field(info.num_chars, config.chars),
                        if filename.as_str() == "-" {
                            "".to_string()
                        } else {
                            format!(" {}", filename)
                        }
                    );

                    total_lines += info.num_lines; ❷
                    total_words += info.num_words;
                    total_bytes += info.num_bytes;
                    total_chars += info.num_chars;
                }
            }
        }
    }

    if config.files.len() > 1 { ❸
        println!(
            "{}{}{}{} total",
            format_field(total_lines, config.lines),
            format_field(total_words, config.words),
            format_field(total_bytes, config.bytes),
            format_field(total_chars, config.chars)
        );
    }
```

```
    Ok(())
}
```

❶ Create mutable variables to track the total number of lines, words, bytes, and characters.

❷ Update the totals using the values from this file.

❸ Print the totals if there is more than one input.

This appears to work well:

```
$ cargo run -- tests/inputs/*.txt
       4      29     177 tests/inputs/atlamal.txt
       0       0       0 tests/inputs/empty.txt
       1       9      48 tests/inputs/fox.txt
       5      38     225 totalcargo run -- -m
tests/inputs/atlamal.txt
```

I can count characters instead of bytes:

```
$ cargo run -- -m tests/inputs/atlamal.txt
     159 tests/inputs/atlamal.txt
```

I can show and hide any columns I want:

```
$ cargo run -- -wc tests/inputs/atlamal.txt
      29     177 tests/inputs/atlamal.txt
```

Most importantly, **`cargo test`** shows all passing tests.

Going Further

Write a version that mimics the output from the GNU `wc` instead of the BSD version. If your system already has the GNU version, run the *mk-outs.sh* program to generate the expected outputs for the given input files. Modify the program to create the correct output according to the tests. Then expand the program to handle the additional options like `--files0-from` for reading

the input filenames from a file and `--max-line-length` to print the length of the longest line. Add tests for the new functionality.

Next, ponder the mysteries of the `iswspace` function mentioned in the BSD manual page noted at the beginning of the chapter. I had wanted to include a test file of the Issa haiku from Chapter 2 but in the original Japanese characters[3]:

隅の蜘案じな煤はとらぬぞよ

—Issa

BSD `wc` thinks there are 3 words:

```
$ wc spiders.txt
       1       3      40 spiders.txt
```

The GNU version says there is only 1 word:

```
$ wc spiders.txt
 1  1 40 spiders.txt
```

I didn't want to open that can of worms, but if you were creating a version of this program to release to the public, what would you report for the number of words?

Summary

Reflect upon your progress in this chapter:

- You learned that the `Iterator::all` function will return `true` if all the elements evaluate to `true` for the given predicate, which is a closure accepting an element. Many similar `Iterator` methods accept a closure as an argument for testing, selecting, and transforming the elements.
- You used the `str::split_whitespace` and `str::chars` methods to break text into words and characters.

- You used the `Iterator::count` method to count the number of items.
- You wrote a function to conditionally format a value or the empty string to support the printing or omission of information according to the flag arguments.
- You organized your unit tests into a `tests` module and imported functions from the parent module called `super`.
- You saw how to use `std::io::Cursor` to create a fake filehandle for testing a function that expects something that implements `BufRead`.
- In about 200 lines of Rust, you wrote a pretty passable replacement for one of the most widely used Unix programs.

---

1 There are many who know how of old did men, In counsel gather; little good did they get; In secret they plotted, it was sore for them later, And for Gjuki's sons, whose trust they deceived.

2 When my youngest first started brushing his own his teeth before bed, I would ask if he'd brushed and flossed. The problem was that he was prone to fibbing, so it was hard to trust him. In an actual exchange one night, I asked "Did you brush and floss your teeth?" *Yes*, he replied. "Did you brush your teeth?" *Yes*, he replied. "Did you floss your teeth?" *No*, he replied. So clearly he failed to properly combine Boolean values because a `true` statement *and* a `false` statement should result in a `false` outcome.

3 A more literal translation might be "Corner spider, rest easy, my soot-broom is idle."

Chapter 6. Den of Uniquity

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 6th Chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the author at kyclark@gmail.com.

There's only one everything

—The Might Be Giants

The `uniq` program (pronounced *unique*) will find the distinct strings in lines of text either from a file or `STDIN`. Among its many uses, it is often employed to count how many times each unique string is found. I think you will find it challenging to create a program that mimics the output from the original tool. Here are some of the skills you will learn:

- How to write to a file or `STDOUT` (pronounced *standard out*)
- How and why to write a closure to capture a variable
- About the don't repeat yourself (DRY) concept
- About the `Write` trait and the `write!` and `writeln!` macros
- How and why to use temporary files
- How to indicate the lifetime of a variable

- How to write tests that pass arguments on the command line or through `STDIN`

How uniq Works

Following is part of the manual page for the BSD version of `uniq`. The challenge program in this chapter will only implement the reading of a file or `STDIN`, writing to a file or `STDOUT`, and counting the lines for the `-c` flag:

```
UNIQ(1)                   BSD General Commands Manual
UNIQ(1)

NAME
     uniq -- report or filter out repeated lines in a file

SYNOPSIS
     uniq [-c | -d | -u] [-i] [-f num] [-s chars] [input_file
[output_file]]

DESCRIPTION
     The uniq utility reads the specified input_file comparing
adjacent lines,
     and writes a copy of each unique input line to the
output_file.  If
     input_file is a single dash ('-') or absent, the standard
input is read.
     If output_file is absent, standard output is used for output.
The second
     and succeeding copies of identical adjacent input lines are
not written.
     Repeated lines in the input will not be detected if they are
not adja-
     cent, so it may be necessary to sort the files first.

     The following options are available:

     -c      Precede each output line with the count of the number
of times
             the line occurred in the input, followed by a single
space.

     -d      Only output lines that are repeated in the input.
```

```
     -f num  Ignore the first num fields in each input line when
doing compar-
             isons.  A field is a string of non-blank characters
separated
             from adjacent fields by blanks.  Field numbers are one
based,
             i.e., the first field is field one.

     -s chars
             Ignore the first chars characters in each input line
when doing
             comparisons.  If specified in conjunction with the -f
option, the
             first chars characters after the first num fields will
be
             ignored.  Character numbers are one based, i.e., the
first char-
             acter is character one.

     -u      Only output lines that are not repeated in the input.

     -i      Case insensitive comparison of lines.
```

In the *06_uniq* directory of the Git repository, you will find various input files I'll use for testing. To start, note that `uniq` will print nothing when given an empty file:

```
$ uniq tests/inputs/empty.txt
```

Given a file with just one line, the one line will be printed:

```
$ uniq tests/inputs/one.txt
a
```

It will also print the number of times a line occurs before the line when run with the `-c` option. The count is right-justified in a field 4 characters wide and is followed by a single space and then the line:

```
$ uniq -c tests/inputs/one.txt
   1 a
```

Given input that contains two duplicate lines, only one line will be emitted:

```
$ cat tests/inputs/two.txt
a
a
$ uniq tests/inputs/two.txt
a
```

The line is shown to have a count of 2:

```
$ uniq -c tests/inputs/two.txt
   2 a
```

A longer input file shows that `uniq` only considers the lines in order and not globally as the line *one* shows several times in this input file:

```
$ cat tests/inputs/three.txt
a
a
b
b
a
c
c
c
a
d
d
d
d
```

When counting, `uniq` starts over at 1 each time it sees a new string:

```
$ uniq -c tests/inputs/three.txt
   2 a
   2 b
   1 a
   3 c
   1 a
   4 d
```

If you wanted *actually* unique values, you must first `sort` the input. Note in the following command that `uniq` will read `STDIN` by default:

```
$ sort tests/inputs/three.txt | uniq -c
   4 a
   2 b
   3 c
   4 d
```

The file *tests/inputs/skip.txt* contains a blank line in the input acts just like any other value and so it will reset the counter:

```
$ uniq -c tests/inputs/skip.txt
   1 a
   1
   1 a
   1 b
```

If you study the usage closely, you'll see a very subtle indication of how to write the output to a file. Notice how `input` and `output` in the following are grouped inside square brackets to indicate that they are optional *as a pair*. That is, if you provide `input`, then you may also optionally provide `output`:

```
usage: uniq [-c | -d | -u] [-i] [-f fields] [-s chars] [input
[output]]
```

For example, I can count *tests/inputs/two.txt* and place the output into *out*:

```
$ uniq -c tests/inputs/two.txt out
$ cat out
      2 a
```

With no positional arguments, `uniq` will read from `STDIN` by default:

```
$ cat tests/inputs/two.txt | uniq -c
      2 a
```

If you want to read from `STDIN` *and* indicate the output filename, you must use a dash ("-" for the input filename:

```
$ cat tests/inputs/two.txt | uniq -c - out
```

```
$ cat out
      2 a
```

The GNU version works basically the same while also providing many more options:

```
$ uniq --help
Usage: uniq [OPTION]... [INPUT [OUTPUT]]
Filter adjacent matching lines from INPUT (or standard input),
writing to OUTPUT (or standard output).

With no options, matching lines are merged to the first occurrence.

Mandatory arguments to long options are mandatory for short options
too.
  -c, --count           prefix lines by the number of occurrences
  -d, --repeated        only print duplicate lines, one for each
group
  -D, --all-repeated[=METHOD]  print all duplicate lines
                          groups can be delimited with an empty
line
                          METHOD={none(default),prepend,separate}
  -f, --skip-fields=N   avoid comparing the first N fields
      --group[=METHOD]  show all items, separating groups with an
empty line
                          METHOD=
{separate(default),prepend,append,both}
  -i, --ignore-case     ignore differences in case when comparing
  -s, --skip-chars=N    avoid comparing the first N characters
  -u, --unique          only print unique lines
  -z, --zero-terminated  end lines with 0 byte, not newline
  -w, --check-chars=N   compare no more than N characters in lines
      --help     display this help and exit
      --version  output version information and exit

A field is a run of blanks (usually spaces and/or TABs), then non-
blank
characters.  Fields are skipped before chars.

Note: 'uniq' does not detect repeated lines unless they are
adjacent.
You may want to sort the input first, or use 'sort -u' without
'uniq'.
Also, comparisons honor the rules specified by 'LC_COLLATE'.
```

As you can see, both the BSD and GNU versions have many more options, but this is as much as the challenge program is expected to implement.

Getting Started

Just so there are no surprises, this program will be called `uniqr` for a Rust version of `uniq`. Start by running **`cargo new uniqr`** to create a new binary package, then modify your *Cargo.toml* to add the following dependencies:

```
[dependencies]
clap = "2.33"

[dev-dependencies]
assert_cmd = "1"
predicates = "1"
tempfile = "3"
rand = "0.8"
```

Copy my *06_uniqr/tests* directory into your projects, and then run **`cargo test`** to ensure the program compiles and that the tests run and fail.

Defining the Arguments

If you would like to copy my normal module structure, then modify your *src/main.rs* to the following:

```
fn main() {
    if let Err(e) = uniqr::get_args().and_then(uniqr::run) {
        eprintln!("{}", e);
        std::process::exit(1);
    }
}
```

You can start your *src/lib.rs* like previous chapters:

```
use clap::{App, Arg};
use std::error::Error;
```

```
type MyResult<T> = Result<T, Box<dyn Error>>;

#[derive(Debug)]
pub struct Config {
    in_file: String, ❶
    out_file: Option<String>, ❷
    count: bool, ❸
}
```

❶ The program will either be read from a given `in_file` or `STDIN`.

❷ The output will either be written to a given `out_file` or `STDOUT`.

❸ The `count` is a Boolean for whether to print the counts of each line.

Here is an outline for `get_args`:

```
pub fn get_args() -> MyResult<Config> {
    let matches = App::new("uniq")
        .version("0.1.0")
        .author("Ken Youens-Clark <kyclark@gmail.com>")
        .about("Rust uniq")
        // What goes here?
        .get_matches();

    Ok(Config {
        in_file: ...
        out_file: ...
        count: ...
    })
}
```

I suggest you start your `run` by printing the `config`:

```
pub fn run(config: Config) -> MyResult<()> {
    println!("{:?}", config);
    Ok(())
}
```

Your program should be able to produce the following usage:

```
$ cargo run -- -h
uniqr 0.1.0
```

```
Ken Youens-Clark <kyclark@gmail.com>
Rust uniq

USAGE:
    uniqr [FLAGS] [ARGS]

FLAGS:
    -c, --count      Show counts ❶
    -h, --help       Prints help information
    -V, --version    Prints version information

ARGS:
    <FILE>    Input file [default: -] ❷
    <FILE>    Output file ❸
```

❶ The `-c|--count` flag is optional.

❷ The input file is the first positional argument and defaults to a dash.

❸ The output file is the second positional argument and is optional.

By default the program will read from `STDIN` which can be represented using a dash:

```
$ cargo run
Config { in_file: "-", out_file: None, count: false }
```

The first positional argument should be interpreted as the `in_file`, the second positional argument as the `out_file`. Note that `clap` can handle options either before or after positional arguments:

```
$ cargo run -- tests/inputs/one.txt out --count
Config { in_file: "tests/inputs/one.txt", out_file: Some("out"),
count: true }
```

TIP

I am trying to mimic the original versions as much as possible, but I do not like optional positional parameters. In my opinion, it would be better to have an `-o|--output` option that defaults to `STDOUT` and have only one optional positional argument for the input file which defaults to `STDIN`.

I assume you are an upright and moral person who figured out how to write that on your own, so I will now share my solution to this:

```
pub fn get_args() -> MyResult<Config> {
    let matches = App::new("uniq")
        .version("0.1.0")
        .author("Ken Youens-Clark <kyclark@gmail.com>")
        .about("Rust uniq")
        .arg(
            Arg::with_name("in_file")
                .value_name("FILE")
                .help("Input file")
                .default_value("-"),
        )
        .arg(
            Arg::with_name("out_file")
                .value_name("FILE")
                .help("Output file"),
        )
        .arg(
            Arg::with_name("count")
                .value_name("COUNT")
                .help("Show counts")
                .short("c")
                .long("count")
                .takes_value(false),
        )
        .get_matches();

    Ok(Config {
        in_file:
matches.value_of("in_file").map(str::to_string).unwrap(), ❶
        out_file: matches.value_of("out_file").map(String::from),
❷
        count: matches.is_present("count"), ❸
    })
}
```

❶ Convert the `in_file` argument to a `String`.

❷ Convert the `out_file` argument to an `Option<String>`.

❸ The `count` is either present or not, so convert to a `bool`.

In the preceding code, `ArgMatches::value_of` method returns

`Option<&str>`. To convert `in_file` to a `String`, I'm calling `Option::map` which would normally take a closure, which I first mentioned in Chapter 5. Instead, I'm using two existing functions that will convert the `&str` to a `String`: `str::to_string`, and `String::from`, which could also be written as `From::from` because Rust infers the `String` type. Remember, functions are first-class objects that can be passed as arguments to other functions.

Because I have defined the `in_file` argument to have a default value, it should never be `None`; therefore, I can use `Option::map` to convert `Some(&str)` to `Some(String)` and do nothing with a `None`. I similarly do this for the `out_file` but leave it as an `Option` since it is not required.

VARIABLE LIFETIMES

You may wonder why I don't leave `in_file` as a `&str` value, and I would encourage you to attempt to change your program like so:

```
#[derive(Debug)]
pub struct Config {
    in_file: &str,
    out_file: Option<&str>,
    count: bool,
}

pub fn get_args() -> MyResult<Config> {
    let matches = App::new("uniq")
        ...

    Ok(Config {
        in_file: matches.value_of("in_file").unwrap(),
        out_file: matches.value_of("out_file"),
        count: matches.is_present("count"),
    })
}
```

Then enjoy your time specifying the lifetimes of these values to the compiler:

```
error[E0106]: missing lifetime specifier
```

```
   --> src/lib.rs:11:14
    |
11  |     in_file: &str,
    |              ^ expected named lifetime parameter
    |
help: consider introducing a named lifetime parameter
    |
10  | pub struct Config<'a> {
11  |     in_file: &'a str,
```

The *lifetime* refers to how long a value is valid for borrowing throughout a program. I would prefer to leave the discussion of lifetimes to other texts such as *Programming Rust* (O'Reilly), and in the next section I'll have reason to use lifetimes. For now, know that it's much easier to return a dynamically and heap-allocated `String` than a `&str` from a function.

Testing the Program

The test suite in *tests/cli.rs* is fairly large containing 78 tests. I intended to test the program under the following conditions:

1. Input file as the only positional argument, check `STDOUT`
2. Input file as a positional argument with `--count` option, check `STDOUT`
3. Input from `STDIN` with no positional arguments, check `STDOUT`
4. Input from `STDIN` with `--count` and no positional arguments, check `STDOUT`
5. Input and output files as positional arguments, check output file
6. Input and output files as positional arguments with `--count`, check output file
7. Input from `STDIN` and output files as positional arguments with `--count`, check output file

Given how large and complicated the tests became, I incorporated a bit more structure in the code. I encourage you to read the tests which start off like so:

```
use assert_cmd::Command;
use predicates::prelude::*;
use rand::{distributions::Alphanumeric, Rng};
use std::fs;
use tempfile::NamedTempFile; ❶

type TestResult = Result<(), Box<dyn std::error::Error>>;

struct Test<'a> { ❷
    input: &'a str,
    out: &'a str,
    out_count: &'a str,
}
```

❶ This is used to create temporary output files.

❷ A struct to define the input files and expected output values with and without the counts.

Note the use of `'a` to denote the lifetime of the values. I want to define structs with `&str` values, and the Rust compiler would like to know exactly how long the values are expected to stick around relative to each other. The fact that all the values have the same lifetime `'a` means that they are all expected to live as long as each other. If you remove the lifetime annotations and run the tests, you'll see similar warnings from the compiler as shown in the previous section along with an suggestion of exactly how to fix it:

```
error[E0106]: missing lifetime specifier
 --> tests/cli.rs:8:12
  |
8 |     input: &str,
  |            ^ expected named lifetime parameter
  |
help: consider introducing a named lifetime parameter
  |
7 | struct Test<'a> {
8 |     input: &'a str,
```

Next, I define the constant values I need for testing:

```
const PRG: &str = "uniqr"; ❶

const EMPTY: Test = Test {
    input: "tests/inputs/empty.txt", ❷
    out: "tests/inputs/empty.txt.out", ❸
    out_count: "tests/inputs/empty.txt.c.out", ❹
};
```

❶ The name of the program being tested

❷ The location of the input file for this test

❸ The location of the output file without the counts

❹ The location of the output file with the counts

After the declaration of `EMPTY`, there are many more `Test` structures followed by several helper functions. The `run` function will use the `Test.input` as an input file and will compare `STDOUT` to the contents of the `Test.out` file:

```
fn run(test: &Test) -> TestResult { ❶
    let expected = fs::read_to_string(test.out)?; ❷
    Command::cargo_bin(PRG)? ❸
        .arg(test.input)
        .assert()
        .success()
        .stdout(expected);
    Ok(())
}
```

❶ The function accepts a `Test` and returns a `TestResult`.

❷ Try to read the expected output file.

❸ Try to run the program with the input file as an argument, verify it ran successfully, and compare the `STDOUT` to the expected value.

The next helper works very similarly but this time tests for the counting:

```
fn run_count(test: &Test) -> TestResult {
    let expected = fs::read_to_string(test.out_count)?; ❶
    Command::cargo_bin(PRG)?
        .args(&[test.input, "-c"]) ❷
        .assert()
        .success()
        .stdout(expected);
    Ok(())
}
```

❶ Read the `Test.out_count` file for the expected output.

❷ Pass both the `Test.input` value and the flag `-c` to count the lines.

The next function will pass the input as `STDIN`:

```
fn run_stdin(test: &Test) -> TestResult {
    let input = fs::read_to_string(test.input)?; ❶
    let expected = fs::read_to_string(test.out)?; ❷
    Command::cargo_bin(PRG)? ❸
        .write_stdin(input)
        .assert()
        .success()
        .stdout(expected);
    Ok(())
}
```

❶ Try to read the `Test.input` file.

❷ Try to read the `Test.out` file.

❸ Pass the `input` through `STDIN` and verify that `STDOUT` is the expected value.

The next function tests both reading from `STDIN` and counting the lines:

```
fn run_stdin_count(test: &Test) -> TestResult {
    let input = fs::read_to_string(test.input)?;
    let expected = fs::read_to_string(test.out_count)?;
    Command::cargo_bin(PRG)? ❶
        .arg("--count")
        .write_stdin(input)
        .assert()
```

```
            .success()
            .stdout(expected);
        Ok(())
    }
```

❶ Run the program with the long `--count` flag and feed the input to `STDIN`, verify that `STDOUT` is correct.

The next function checks that the program accepts both the input and output files as positional arguments. This is somewhat more interesting as I needed to use temporary files in the testing because, as you have seen repeatedly, Rust will run the tests in parallel. If I were to use the same dummy filename like *blargh* to write all the output files, the tests would overwrite each other's output. To get around this, I use the `tempfile::NamedTempFile` to get a dynamically generated temporary filename that will automatically be removed when I finish:

```
fn run_outfile(test: &Test) -> TestResult {
    let expected = fs::read_to_string(test.out)?;
    let outfile = NamedTempFile::new()?; ❶
    let outpath = &outfile.path().to_str().unwrap(); ❷

    Command::cargo_bin(PRG)? ❸
        .args(&[test.input, outpath])
        .assert()
        .success()
        .stdout("");
    let contents = fs::read_to_string(&outpath)?; ❹
    assert_eq!(&expected, &contents); ❺

    Ok(())
}
```

❶ Try to get a named temporary file.

❷ Get the `path` to the file.

❸ Run the program with the input and output filenames as arguments, verify there is nothing in `STDOUT`.

❹ Try to read the output file.

❺ Check that the contents of the output file match the expected value.

The next two functions are variations on what I've already shown, adding in the `--count` flag and finally asking the program to read from `STDIN` when the input filename is a dash. The rest of the module calls these helpers using the various structs to run all the tests. It was tedious to write all this, but it's very important to test every single aspect of a program with all possible combinations of the arguments. You can perhaps see now why I didn't want to add any more functionality to the challenge program.

One interesting aspect of implementing open-source programs like `uniq` is that you can sometimes find the source code for those programs. While looking for the original test suite, I found a Perl program used to test the GNU version. I used several of those examples in the *tests/inputs/t[1-6].txt* input files, as you can see in *06_uniqr/mk-outs.sh*:

```
$ cat mk-outs.sh
#!/usr/bin/env bash

ROOT="tests/inputs"
OUT_DIR="tests/expected"

[[ ! -d "$OUT_DIR" ]] && mkdir -p "$OUT_DIR"

# Cf
https://github.com/coreutils/coreutils/blob/master/tests/misc/uniq.
pl
echo -ne "a\na\n"    > $ROOT/t1.txt ❶
echo -ne "a\na"      > $ROOT/t2.txt ❷
echo -ne "a\nb"      > $ROOT/t3.txt ❸
echo -ne "a\na\nb"   > $ROOT/t4.txt ❹
echo -ne "b\na\na\n" > $ROOT/t5.txt ❺
echo -ne "a\nb\nc\n" > $ROOT/t6.txt ❻

for FILE in $ROOT/*.txt; do
    BASENAME=$(basename "$FILE")
    uniq      $FILE > ${OUT_DIR}/${BASENAME}.out
    uniq -c   $FILE > ${OUT_DIR}/${BASENAME}.c.out
    uniq    < $FILE > ${OUT_DIR}/${BASENAME}.stdin.out
    uniq -c < $FILE > ${OUT_DIR}/${BASENAME}.stdin.c.out
done
```

❶ Two lines each ending with a newline

❷ No trailing newline on last line

❸ Two different lines, no trailing newline

❹ Two lines the same, last is different with no trailing newline

❺ Two different values with newlines on each

❻ Three different values with newlines on each

I found it surprisingly challenging to exactly match the output from `uniq`, but I guess I'm a better person for the effort now. I would suggest start in *src/lib.rs* by reading the input file, and again it makes sense to use the `open` function from previous chapters:

```
fn open(filename: &str) -> MyResult<Box<dyn BufRead>> {
    match filename {
        "-" => Ok(Box::new(BufReader::new(io::stdin()))),
        _ => Ok(Box::new(BufReader::new(File::open(filename)?))),
    }
}
```

Be sure you expand your imports to include the following:

```
use clap::{App, Arg};
use std::{ ❶
    error::Error,
    fs::File,
    io::{self, BufRead, BufReader},
};
```

❶ This syntax lets me group imports by common prefixes, so all the following come from `std`.

You can borrow quite a bit of code from Chapter 3 that reads lines of text from an input file or `STDIN` while preserving the line endings:

```
pub fn run(config: Config) -> MyResult<()> {
    let mut file = open(&config.in_file)
```

```
            .map_err(|e| format!("{}: {}", config.in_file, e))?; ❶
        let mut line = String::new(); ❷
        loop { ❸
            let bytes = file.read_line(&mut line)?; ❹
            if bytes == 0 { ❺
                break;
            }
            print!("{}", line); ❻
            line.clear(); ❼
        }
        Ok(())
    }
```

❶ Read either `STDIN` if the input file is a dash or open the given filename. Create an informative error message when this fails.

❷ Create a new, empty mutable `String` buffer to hold each line.

❸ Create an infinite loop.

❹ Read a line of text while preserving the line endings.

❺ If no bytes were read, break out of the loop.

❻ Print the line buffer.

❼ Clear the line buffer.

Run your program with an input file to ensure it works:

```
$ cargo run -- tests/inputs/one.txt
a
```

It should also work for reading `STDIN`:

```
$ cargo run -- - < tests/inputs/one.txt
a
```

Try to get your program to iterate the lines of input and count each unique run of lines, then try to get your program to print the lines with and without the counts. Look at how `open` works, and try to copy those ideas to use `File::create` to write output to a file or `STDOUT`. Remember that you can run just a subset of tests with a command like **`cargo test empty`** to

run all the tests with the string *empty* in the name. I know you can do this.

Solution

I'll step you through how I arrived at a solution. Your version may be different, but it's fine as long as it passes the test suite. Building on the last version I showed, I decided to create two additional mutable variables to hold the last line of text and the running count. For now, I will always print the count to make sure it's working correctly:

```
pub fn run(config: Config) -> MyResult<()> {
    let mut file = open(&config.in_file)
        .map_err(|e| format!("{}: {}", config.in_file, e))?;
    let mut line = String::new();
    let mut last = String::new(); ❶
    let mut count: u64 = 0; ❷

    loop {
        let bytes = file.read_line(&mut line)?;
        if bytes == 0 {
            break;
        }

        if line.trim_end() != last.trim_end() { ❸
            if count > 0 { ❹
                print!("{:>4} {}", count, last); ❺
            }
            last = line.clone(); ❻
            count = 0; ❼
        }

        count += 1; ❽
        line.clear();
    }

    if count > 0 { ❾
        print!("{:>4} {}", count, last);
    }

    Ok(())
}
```

❶ Create a mutable variable to hold the last line of text.

❷ Create a mutable variable to hold the count.

❸ Compare the current line to the last time, both trimmed of any possible trailing whitespace.

❹ Only print the output when `count` is greater than 0.

❺ Print the `count` right-justified in a column 4 characters wide followed by a space and the `line` value.

❻ Set the `last` variable to a copy of the current `line`.

❼ Reset the counter to 0.

❽ Increment the counter by 1.

❾ Handle the last line of the file.

If I run **`cargo test`**, this will pass a goodly number of tests. This code is clunky, though. I don't like having to check `if count > 0` twice as it violates the *don't repeat yourself* (DRY) principle, so I'm thinking that needs to be put into a function. I'm also always printing the count, but I should only print this when `config.count` is `true`—another condition that could go into this function. I decided to write this as a closure inside the `run` function to *close around* the `config.count` value:

```
let print = |count: &u64, line: &String| { ❶
    if count > &0 { ❷
        if config.count { ❸
            print!("{:>4} {}", &count, &line); ❹
        } else {
            print!("{}", &line); ❺
        }
    };
};
```

❶ The `print` closure will accept `count` and `line` values.

❷ Only print if the `count` is greater than 0.

❸ Check if the `config.count` value is `true`.

❹ Use the `print!` macro to print the `count` and `line` to `STDOUT`.

❺ Otherwise, print the `line` to `STDOUT`.

CLOSURES VERSUS FUNCTIONS

A closure is a function, so you might be tempted to write `print` as a function inside the `run` function:

```
pub fn run(config: Config) -> MyResult<()> {
    ...
    fn print(count: &u64, line: &String) {
        if count > &0 {
            if config.count {
                print!("{:>4} {}", &count, &line);
            } else {
                print!("{}", &line);
            }
        };
    };
    ...
```

This is a common way to write a closure in other languages, and Rust does allow you to declare a function inside another function; however, the Rust compiler specifically disallows capturing a dynamic value from the environment[1]:

```
error[E0434]: can't capture dynamic environment in a fn item
  --> src/lib.rs:67:16
   |
67 |             if config.count {
   |                ^^^^^^
   |
   = help: use the `|| { ... }` closure form instead
```

I can update the rest of the function to use this closure:

```
loop {
    let bytes = file.read_line(&mut line)?;
    if bytes == 0 {
        break;
```

```
        }

        if line.trim_end() != last.trim_end() {
            print(&count, &last);
            last = line.clone();
            count = 0;
        }

        count += 1;
        line.clear();
    }

    print(&count, &last);
```

With these changes, my program will pass several more tests. If I look at the failed tests, they all contain *outfile* because I'm failing to write a named output file. I can add this last feature with two changes. First, I would like to open the output file in the same way as I open the input file by either creating a named output file or by using `STDOUT`. You'll need to add `use std::io::Write` for the following code, which you can place just just after the `file` variable:

```
let mut out_file: Box<dyn Write> = match &config.out_file { ❶
    Some(out_name) => Box::new(File::create(&out_name)?), ❷
    _ => Box::new(io::stdout()), ❸
};
```

❶ The mutable `out_file` will be a boxed value that implements the `std::io::Write` trait.

❷ When `config.out_file` is `Some` filename, use `File::create` to try to create the file.

❸ Otherwise, use `std::io::stdout`.

If you follow the documentation for both `File::create` and `io::stdout`, you'll see both have a *Traits* section showing the various traits they implement. Both show that they implement `Write`, and so they satisfy the type requirement `Box<dyn Write>` which says that the value inside the `Box` must implement this trait.

The second change I need to make is to use `out_file` for the output. I will replace the `print!` macro with `write!` to write the output to a stream like a filehandle or `STDOUT`. The first argument to `write!` must be a mutable value that implements the `Write` trait. The documentation shows that `write!` will return a `std::io::Result` because it might fail. As such, I changed my `print` closure to return `MyResult`. Here is the final version of my `run` function that passes all the tests:

```
pub fn run(config: Config) -> MyResult<()> {
    let mut file = open(&config.in_file)
        .map_err(|e| format!("{}: {}", config.in_file, e))?; ❶
    let mut out_file: Box<dyn Write> = match &config.out_file { ❷
        Some(out_name) => Box::new(File::create(&out_name)?),
        _ => Box::new(io::stdout()),
    };

    let mut print = |count: &u64, line: &String| -> MyResult<()> {
❸
        if count > &0 {
            if config.count {
                write!(out_file, "{:>4} {}", &count, &line)?;
            } else {
                write!(out_file, "{}", &line)?;
            }
        };
        Ok(())
    };

    let mut line = String::new();
    let mut last = String::new();
    let mut count: u64 = 0;
    loop {
        let bytes = file.read_line(&mut line)?;
        if bytes == 0 {
            break;
        }

        if line.trim_end() != last.trim_end() {
            print(&count, &last)?; ❹
            last = line.clone();
            count = 0;
        }

        count += 1;
```

```
            line.clear();
        }

        print(&count, &last)?; ❺

        Ok(())
    }
```

❶ Open either `STDIN` or the given input filename.

❷ Open either `STDOUT` or the given output filename.

❸ Create a mutable `print` closure to format the output.

❹ Use the `print` closure to possibly print output. Use `?` to propagate potential errors.

❺ Handle the last line of the file.

I would think it's highly unlikely that you would independently write something exactly like my solution. As long as your solution passes the tests, it's perfectly acceptable. Part of what I like about writing with tests is that there is an objective determination when a program meets some level of specifications. Louis Srygley says, "Without requirements or design, programming is the art of adding bugs to an empty text file." I would say that tests are the requirements made incarnate. Without tests, you simply have no way to know when a change to your program strays from the requirements or breaks the design.

In closing, I would share that I explored a couple of other ways to write this algorithm, one of which read all the lines of the input file into a vector and looked at pairs of lines using `Vec::windows`. This was interesting but could fail if the size of the input file exceeded the available memory on my machine. The solution presented here will only ever allocate memory for the current and last lines and so should scale to any size file.

Going Further

As usual, the BSD and GNU versions of `uniq` both have many more features

than I chose to include in the challenge. I would encourage you to add all the features you would like to have in your version. Consider reading the original source code and tests to see what you can borrow. Be sure to add tests for each feature, and always run the entire test suite to verify that all previous features still work.

In my mind, `uniq` is closely tied with `sort` as I often use them together. Consider implementing your own version of `sort`, at least to the point of sorting values lexicographically (in dictionary order) or numerically.

Summary

I hope you found this challenge as interesting to write as I did. Let's review some of the things you learned:

- You can now open a new file for writing or use `STDOUT`.
- The idea of DRY is that you move any duplicated code into a single abstraction like a function or a closure.
- You've seen that a closure can be used to capture values from the enclosing scope.
- You learned about the `Write` trait and how values that implement this trait can be used with the `write!` and `writeln!` macros.
- Sometimes you will find you need to write a temporary file. You've now used a Rust crate that helps find a unique filename and open a filehandle for you.
- The Rust compiler may sometimes require you to indicate the lifetime of a variable which is how long it lives in relation to other variables.

---

1 The name *closure* refers to this capturing or *closing around* a lexically scoped binding.

Chapter 7. Finders Keepers

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 7th Chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the author at kyclark@gmail.com.

> *Then is when I maybe should have wrote it down but when I looked around to find a pen and then I tried to think of what you said*
>
> —They Might Be Giants

The `find` utility will, unsurprisingly, find things. With no options, it will recursively search the current working directory for all entries, including files, symbolic links, sockets, directories, and more. You can restrict it to search one or more paths and include restrictions to find entries matching myriad restrictions such as names, file sizes, file types, modification times, permissions, and more. The challenge program you write will locate files, directories, or links in one or more directories having names that match one or more regular expressions.

You will learn:

- How to use `clap` to constrain possible values for command-line arguments

- How to anchor a regular expression to the end of a string
- How to create an enumerated type (`enum`)
- How to recursively search file paths using the `walkdir` crate
- How to use `Iterator::any`
- How to chain multiple `filter` and `map` operations

How find Works

The manual page for `find` is truly amazing. It goes on for about 500 lines detailing all the options you can use to find files and directories. Here is just the beginning that shows the general vibe of the BSD `find`:

```
FIND(1)                   BSD General Commands Manual
FIND(1)

NAME
     find -- walk a file hierarchy

SYNOPSIS
     find [-H | -L | -P] [-EXdsx] [-f path] path ... [expression]
     find [-H | -L | -P] [-EXdsx] -f path [path ...] [expression]

DESCRIPTION
     The find utility recursively descends the directory tree for
each path
     listed, evaluating an expression (composed of the
''primaries'' and
     ''operands'' listed below) in terms of each file in the tree.
```

The GNU `find` is similar:

```
$ find --help
Usage: find [-H] [-L] [-P] [-Olevel]
[-D help|tree|search|stat|rates|opt|exec] [path...] [expression]

default path is the current directory; default expression is -print
expression may consist of: operators, options, tests, and actions:
```

```
operators (decreasing precedence; -and is implicit where no others
are given):
      ( EXPR )   ! EXPR   -not EXPR   EXPR1 -a EXPR2   EXPR1 -and
EXPR2
      EXPR1 -o EXPR2   EXPR1 -or EXPR2   EXPR1 , EXPR2

positional options (always true): -daystart -follow -regextype

normal options (always true, specified before other expressions):
      -depth --help -maxdepth LEVELS -mindepth LEVELS -mount -
noleaf
      --version -xautofs -xdev -ignore_readdir_race -
noignore_readdir_race

tests (N can be +N or -N or N): -amin N -anewer FILE -atime N -cmin
N
      -cnewer FILE -ctime N -empty -false -fstype TYPE -gid N -
group NAME
      -ilname PATTERN -iname PATTERN -inum N -iwholename PATTERN
      -iregex PATTERN -links N -lname PATTERN -mmin N -mtime N
      -name PATTERN -newer FILE -nouser -nogroup -path PATTERN
      -perm [-/]MODE -regex PATTERN -readable -writable -executable
      -wholename PATTERN -size N[bcwkMG] -true -type [bcdpflsD] -
uid N
      -used N -user NAME -xtype [bcdpfls] -context CONTEXT

actions: -delete -print0 -printf FORMAT -fprintf FILE FORMAT -print
      -fprint0 FILE -fprint FILE -ls -fls FILE -prune -quit
      -exec COMMAND ; -exec COMMAND {} + -ok COMMAND ;
      -execdir COMMAND ; -execdir COMMAND {} + -okdir COMMAND ;
```

As usual, the challenge program will only attempt to implement a subset of these options that I'll demonstrate forthwith using the files in *07_findr/tests/inputs*. The following output from `tree` shows the directory and file structure. Note that `->` indicates that *d/b.csv* is a link to the file *a/b/b.csv*. A *link* is a pointer or a shortcut to another file or directory:

```
$ cd 07_findr/tests/inputs
$ tree
.
├── a
│   ├── a.txt
│   └── b
│       ├── b.csv
│       └── c
```

```
│           └── c.mp3
├── d
│   ├── b.csv -> ../a/b/b.csv
│   ├── d.tsv
│   ├── d.txt
│   └── e
│       └── e.mp3
├── f
│   └── f.txt
└── g.csv

6 directories, 9 files
```

> **NOTE**
>
> Windows does not have a symbolic link (AKA *symlink*) like Unix, so there are four tests that will fail because the path *tests\inputs\d\b.csv* exists as a regular file and not as a link. I recommend Windows users explore writing and testing this program in Windows Subsystem for Linux.

To start, `find` will accept one or more positional arguments which are the starting paths. For each path, `find` will recursively search for all files and directories found therein. If I am in the *tests/inputs* directory and indicate `.` for the current working directory, `find` will list all the contents. Note that I will be showing the output from `find` when run on macOS which differs from the ordering of the entries shown on Linux:

```
$ find .
.
./g.csv
./a
./a/a.txt
./a/b
./a/b/b.csv
./a/b/c
./a/b/c/c.mp3
./f
./f/f.txt
./d
./d/b.csv
./d/d.txt
./d/d.tsv
./d/e
```

```
./d/e/e.mp3
```

Using the `-type` option[1], I can specify "f" to only find *files*:

```
$ find . -type f
./g.csv
./a/a.txt
./a/b/b.csv
./a/b/c/c.mp3
./f/f.txt
./d/d.txt
./d/d.tsv
./d/e/e.mp3
```

I can use "l" to only find *links*:

```
$ find . -type l
./d/b.csv
```

I can also use "d" to only find *directories*:

```
$ find . -type d
.
./a
./a/b
./a/b/c
./f
./d
./d/e
```

While the challenge program will only try to find these types, `find` will accept several more `-type` values per the manual page:

```
-type t
     True if the file is of the specified type.  Possible file
types
     are as follows:

     b       block special
     c       character special
     d       directory
     f       regular file
```

```
l       symbolic link
p       FIFO
s       socket
```

If you give a `-type` value not found in this list, `find` will stop with an error:

```
$ find . -type x
find: -type: x: unknown type
```

The `-name` option can locate items matching a file glob pattern such as *.*csv* for any entry ending with *.csv*. The `*` on the command line must be escaped with a backslash so that it is passed as a literal character and not interpreted by the shell:

```
$ find . -name \*.csv
./g.csv
./a/b/b.csv
./d/b.csv
```

You can also put the pattern in quotes:

```
$ find . -name "*.csv"
./g.csv
./a/b/b.csv
./d/b.csv
```

I can search for multiple `-name` patterns by chaining them with `-o` for *or*:

```
$ find . -name "*.txt" -o -name "*.csv"
./g.csv
./a/a.txt
./a/b/b.csv
./f/f.txt
./d/b.csv
./d/d.txt
```

I can combine `-type` and `-name` options. For instance, I can search for files or links matching `*.csv`:

```
$ find . -name "*.csv" -type f -o -type l
./g.csv
./a/b/b.csv
./d/b.csv
```

I must use parentheses to group the `-type` arguments when the `-name` condition follows an *or* expression:

```
$ find . \( -type f -o -type l \) -name "*.csv"
./g.csv
./a/b/b.csv
./d/b.csv
```

I can also list multiple search paths as positional arguments:

```
$ find a/b d -name "*.mp3"
a/b/c/c.mp3
d/e/e.mp3
```

If the given search path is an invalid directory, `find` will print an error:

```
$ find blargh
find: blargh: No such file or directory
```

I find it odd that `find` accepts files as a path argument, simply printing the filename:

```
$ find a/a.txt
a/a.txt
```

The challenge program, however, will only accept readable directory names as valid arguments. While `find` can do much more, this is as much as you will implement in this chapter.

Getting Started

The program you write will be called `findr` (pronounced *find-er*), and I recommend you run **`cargo new findr`** to start. Update *Cargo.toml* with

the following:

```
[dependencies]
clap = "2.33"
walkdir = "2"
regex = "1"

[dev-dependencies]
assert_cmd = "1"
predicates = "1"
rand = "0.8"
sys-info = "0.9" ❶
```

❶ This module is needed to detect when the tests are running on Windows and make changes in the expected output.

Normally I would suggest that you copy the *07_findr/tests* directory into your project, but this will not work because the symlink in the *tests/inputs* directory will not be preserved causing your tests to fail. Instead, I've provided a `bash` script in the *07_findr* directory that will copy the *tests* into a destination directory. Run with no arguments to see the usage:

```
$ ./cp-tests.sh
Usage: cp-tests.sh DEST_DIR
```

Assuming I created my new project in `$HOME/rust/findr`, I can use the program like this:

```
$ ./cp-tests.sh ~/work/rust/findr
Copying "tests" to "/Users/kyclark/work/rust/findr"
Fixing symlink
Done.
```

Run **`cargo test`** to build the program and run the tests, all of which should fail.

Defining the Arguments

I will use the following for *src/main.rs*:

```
fn main() {
    if let Err(e) = findr::get_args().and_then(findr::run) {
        eprintln!("{}", e);
        std::process::exit(1);
    }
}
```

Before I show you how I started my *src/lib.rs*, I want to show the expected command-line interface:

```
$ cargo run -- --help
findr 0.1.0
Ken Youens-Clark <kyclark@gmail.com>
Rust find

USAGE:
    findr [OPTIONS] [--] [DIR]... ❶

FLAGS:
    -h, --help       Prints help information
    -V, --version    Prints version information

OPTIONS:
    -n, --name <NAME>...    Name ❷
    -t, --type <TYPE>...    Entry type [possible values: f, d, l]
❸

ARGS:
    <DIR>...    Search directory [default: .] ❹
```

❶ The `--` separates multiple optional values from the multiple positional values. Alternatively, you can place the positional arguments before the options as the `find` program does.

❷ The `-n|--name` option can specify one or more patterns.

❸ The `-t|--type` option can specify one or more of *f* for files, *d* for directories, or *l* for links.

❹ TK

You can model this however you like, but here is how I decided to start:

```
use crate::EntryType::*; ❶
use clap::{App, Arg};
use regex::Regex;
use std::error::Error;

type MyResult<T> = Result<T, Box<dyn Error>>;

#[derive(Debug, PartialEq)] ❷
enum EntryType {
    Dir,
    File,
    Link,
}

#[derive(Debug)]
pub struct Config {
    dirs: Vec<String>, ❸
    names: Option<Vec<Regex>>, ❹
    entry_types: Option<Vec<EntryType>>, ❺
}
```

❶ This will allow me to use, for instance, `Dir` instead of `EntryType::Dir`.

❷ The `EntryType` is an enumerated list of possible values.

❸ The `dirs` will be a vector of strings.

❹ The `names` will be an optional vector of compiled regular expressions.

❺ The `entry_types` will be an optional vector of `EntryType` variants.

In the preceding code, I'm introducing `enum`, which is a "type that can be any one of several variants." You've already been using enums such as `Option`, which has the variants `Some<T>` or `None`, and `Result`, which has the variants `Ok<T>` and `Err<E>`. In a language without such a type, you'd probably have to use literal strings in your code like "dir," "file," and "link." In Rust, I can create a new `enum` called `EntryType` with exactly three possibilities: `Dir`, `File`, or `Link`. I can use these values in pattern matching with much more precision than matching strings, which might be misspelled. Additionally, Rust will not allow me to `match` on `EntryType` values without considering all the variants, which adds yet another layer of

safety in using them.

> **TIP**
>
> Per Rust naming conventions, types, structs, traits, and enum variants use `UpperCamelCase`.

Here is how you might start the `get_args` function:

```
pub fn get_args() -> MyResult<Config> {
    let matches = App::new("findr")
        .version("0.1.0")
        .author("Ken Youens-Clark <kyclark@gmail.com>")
        .about("Rust find")
        // What goes here?
        .matches()

    Ok(Config {
        dirs: ...
        names: ...
        entry_types: ...
    })
}
```

Perhaps start the `run` function by printing the `config`:

```
pub fn run(config: Config) -> MyResult<()> {
    println!("{:?}", config);
    Ok(())
}
```

When run with no arguments, the default `Config` values should look like this:

```
$ cargo run
Config { dirs: ["."], names: None, entry_types: None }
```

When given a `--type` argument of "f," the `entry_types` should include the `File` variant:

```
$ cargo run -- --type f
Config { dirs: ["."], names: None, entry_types: Some([File]) }
```

or `Dir` when the value is “d”:

```
$ cargo run -- --type d
Config { dirs: ["."], names: None, entry_types: Some([Dir]) }
```

or `Link` when the value is “l”:

```
$ cargo run -- --type l
Config { dirs: ["."], names: None, entry_types: Some([Link]) }
```

Any other value should be rejected. You can get `clap::Arg` to handle this, so read the documentation closely:

```
$ cargo run -- --type x
error: 'x' isn't a valid value for '--type <TYPE>...'
        [possible values: d, f, l]

USAGE:
    findr --type <TYPE>

For more information try --help
```

I’ll be using the `Regex` module to match file and directory names, which means that the `--name` value must be a valid regular expression. Regex syntax differs slightly from file glob patterns as shown in Figure 6-1. For instance, the asterisk (`*`) in the file glob `*.txt` means *zero or more of any character* and the dot has no special meaning[2], so this will match files that end in *.txt*. In regex syntax, however, the asterisk means *zero or more of the previous character*, so I need to write `.*` where the dot (`.`) is a metacharacter that means *any one character*.

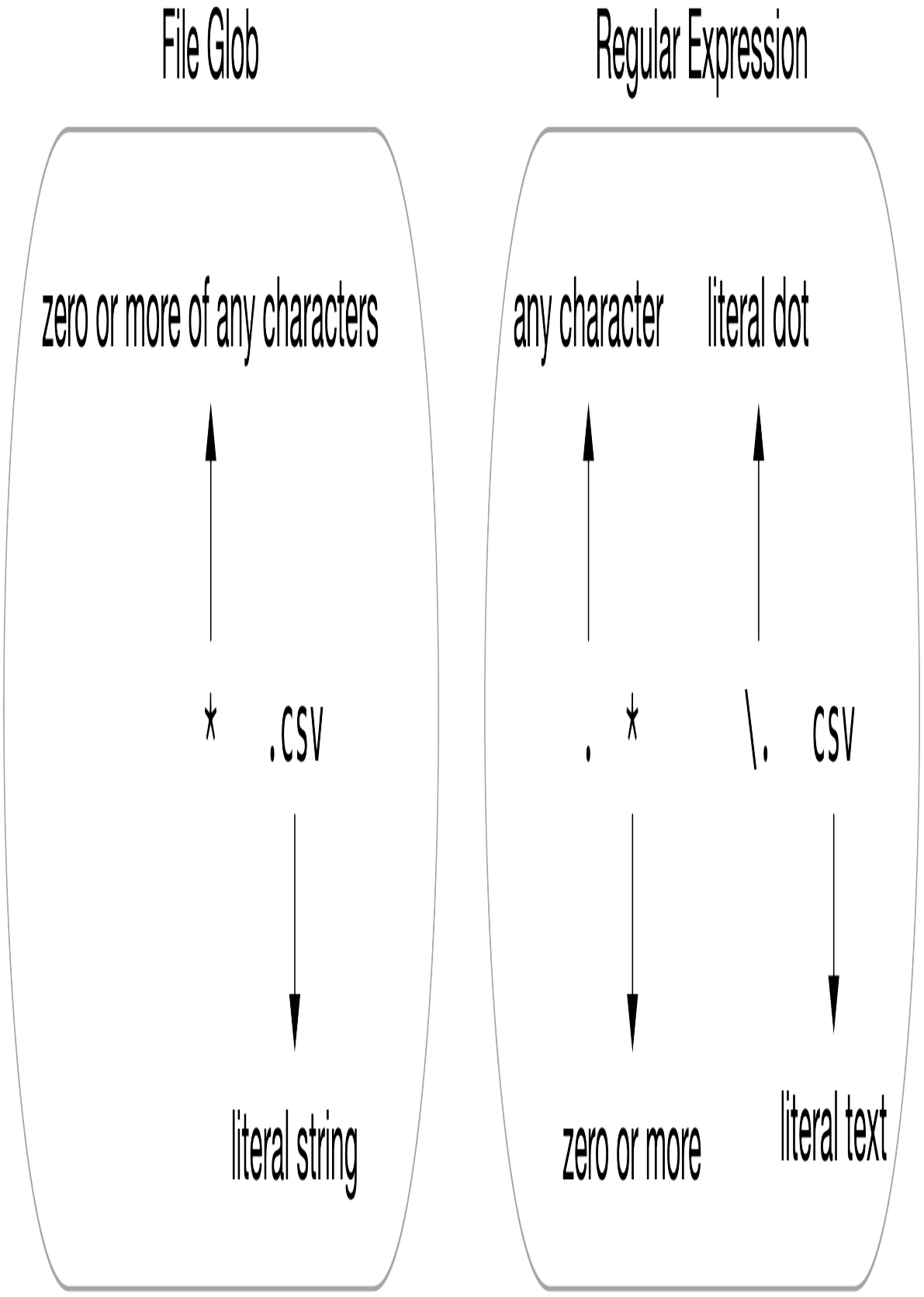

File Glob
zero or more of any characters
*
.csv
literal string
Regular Expression
any character
literal dot
.
*
\.
csv
zero or more
literal text

Figure 7-1. The asterisk `*` and dot `.` have different meanings in file globs versus regular expressions

This means that the equivalent regex should use a backslash to escape the literal dot such as `.*\.txt`, which must be double-escaped on the command line:

```
$ cargo run -- --name .*\\.txt
Config { dirs: ["."], names: Some([.*\.txt]), entry_types: None }
```

Alternatively, you can place the dot inside a character class like `[.]` where it is no longer a metacharacter:

```
$ cargo run -- --name .*[.]txt
Config { dirs: ["."], names: Some([.*[.]txt]), entry_types: None }
```

Technically, the regular expression will match anywhere in the string, even at the beginning because `.*` means zero or more of anything:

```
let re = Regex::new(".*[.]csv").unwrap();
assert!(re.is_match("foo.csv"));
assert!(re.is_match(".csv.foo"));
```

If I want to insist that the regex matches at the end of the string, I can add `$` at the end of the pattern to indicate the end of the string:

```
let re = Regex::new(".*[.]csv$").unwrap();
assert!(re.is_match("foo.csv"));
assert!(!re.is_match(".csv.foo"));
```

TIP

The converse of using `$` to anchor a pattern to the end of a string is to use `^` to indicate the beginning of the string.

If I try to use the same file glob pattern that `find` expects, it will be rejected:

```
$ cargo run -- --name \*.txt
```

```
Invalid --name "*.txt"
```

All the `Config` fields should accept multiple values. For this output, I changed `run` to pretty-print the `config`:

```
$ cargo run -- -t f l -n txt mp3 -- tests/inputs/a tests/inputs/d
Config {
    dirs: [
        "tests/inputs/a",
        "tests/inputs/d",
    ],
    names: Some(
        [
            txt,
            mp3,
        ],
    ),
    entry_types: Some(
        [
            File,
            Link,
        ],
    ),
}
```

It's important to get this much working before attempting to solve the rest of the program. Don't proceed until your program can replicate the preceding output and can pass at least **`cargo test dies`**:

```
running 2 tests
test dies_bad_type ... ok
test dies_bad_name ... ok
```

Validating the Arguments

Following is my `get_args` function so that we can regroup on the task at hand:

```
pub fn get_args() -> MyResult<Config> {
    let matches = App::new("findr")
        .version("0.1.0")
        .author("Ken Youens-Clark <kyclark@gmail.com>")
```

```
        .about("Rust find")
        .arg(
            Arg::with_name("dirs") ❶
                .value_name("DIR")
                .help("Search directory")
                .default_value(".")
                .min_values(1),
        )
        .arg(
            Arg::with_name("names") ❷
                .value_name("NAME")
                .help("Name")
                .short("n")
                .long("name")
                .takes_value(true)
                .multiple(true),
        )
        .arg(
            Arg::with_name("types") ❸
                .value_name("TYPE")
                .help("Entry type")
                .short("t")
                .long("type")
                .possible_values(&["f", "d", "l"])
                .takes_value(true)
                .multiple(true),
        )
        .get_matches();
```

❶ The `dirs` argument requires at least one value and defaults to a dot (`.`).

❷ The `names` option accepts zero or more values.

❸ The `types` option accepts zero or more values of *f*, *d*, or *l*.

Next, I handle the possible filenames, transforming them into regular expressions or rejecting invalid patterns:

```
    let mut names = vec![]; ❶
    if let Some(vals) = matches.values_of_lossy("names") { ❷
        for name in vals { ❸
            match Regex::new(&name) { ❹
                Ok(re) => names.push(re), ❺
                _ => {
                    return Err(From::from(format!( ❻
```

```
                        "Invalid --name \"{}\"",
                        name
                    )))
                }
            }
        }
    }
```

❶ Create a mutable vector to hold the regular expressions.

❷ See if the user has provided `Some(vals)` for the option.

❸ Iterate over the values.

❹ Try to create a new `Regex` with the name.

❺ Add a valid regex to the list of `names`.

❻ Return an error that the pattern is not valid.

Next, I interpret the entry types. Even though I used `Arg::possible_values` to ensure that the user could only supply "f," "d," or "l," Rust still requires a `match` arm for any other possible string:

```
    let entry_types = matches.values_of_lossy("types").map(|vals| { ❶
        vals.iter() ❷
            .filter_map(|val| match val.as_str() { ❸
                "d" => Some(Dir), ❹
                "f" => Some(File),
                "l" => Some(Link),
                _ => None, ❺
            })
            .collect() ❻
    });
```

❶ `ArgMatches.values_of_lossy` will return an `Option<Vec<String>>`. Use `Option::map` to handle `Some(vals)`.

❷ Iterate over each of the values.

❸ Use `Iterator::filter_map` that "yields only the values for which the supplied closure returns `Some(value)`."

❹ If the value is “d,” “f,” or “l,” return the appropriate `EntryType`.

❺ This arm should never be selected, but return `None` anyway.

❻ Use `Iterator::collect` to gather the values into a vector of `EntryType` values.

I end the function by returning the `Config`:

```
        Ok(Config {
            dirs: matches.values_of_lossy("dirs").unwrap(),
            names: if names.is_empty() { None } else { Some(names) }, ❶
            entry_types,
        })
    }
```

❶ The `names` should be `Some` value when present or `None` when absent.

Find All the Things

Now that you have validated the arguments from the user, it’s time to look for the items that match the conditions. You might start by iterating over `config.dirs` and trying to find all the files contained in each. I will use the `walkdir` crate for this. Following is how I can use some of the example code from the documentation to print all the entries. Be sure to add `use walkdir::WalkDir` for the following:

```
pub fn run(config: Config) -> MyResult<()> {
    for dirname in config.dirs {
        for entry in WalkDir::new(dirname) {
            println!("{}", entry?.path().display()); ❶
        }
    }
    Ok(())
}
```

❶ Note the use of `entry?` to unpack the `Result` and propagate an error.

To see if this works, I'll list the contents of the *tests/inputs/a/b*. Note that this is the order I see on macOS:

```
$ cargo run -- tests/inputs/a/b
tests/inputs/a/b
tests/inputs/a/b/b.csv
tests/inputs/a/b/c
tests/inputs/a/b/c/c.mp3
```

On Linux, I see the following output:

```
$ cargo run -- tests/inputs/a/b
tests/inputs/a/b
tests/inputs/a/b/c
tests/inputs/a/b/c/c.mp3
tests/inputs/a/b/b.csv
```

On Windows/Powershell, I see this output:

```
> cargo run -- tests/inputs/a/b
tests/inputs/a/b
tests/inputs/a/b\b.csv
tests/inputs/a/b\c
tests/inputs/a/b\c\c.mp3
```

I've written the test suite to check the lines of output irrespective of order, and I've also included specific output files for Windows to ensure the backslashes are correct. A quick check with **`cargo test`** shows that this simple version of the program already passes several tests. One problem is that this program fails rather ungracefully with a nonexistent directory name causing the program to stop as soon as it tries to read the bad directory:

```
$ cargo run -- blargh tests/inputs/a/b
IO error for operation on blargh: No such file or directory (os
error 2)
```

I recommend you build from this. First, figure out if the given argument names a directory that can be read. If not, print an error to STDERR and move to the next argument. Then iterate over the contents of the directory and show

files, directories, or links when `config.entry_types` contains the appropriate `EntryType`. Next, filter out entry names that fail to match any of the given regular expressions when they are present. I would encourage you to look at the *mk-outs.sh* program I used to generate the expected output files for various executions of the original `find` command, and then read *tests/cli.rs* to see how these commands are translated to work with `findr`.

You got this. I know you can do it.

Solution

As suggested, my first step is to weed out anything that isn't a directory or which can't be read, perhaps due to permission problems. With the following code, the program passes **`cargo test skips_bad_dir`**:

```
pub fn run(config: Config) -> MyResult<()> {
    for dirname in config.dirs {
        match fs::read_dir(&dirname) { ❶
            Err(e) => eprintln!("{}: {}", dirname, e), ❷
            _ => {
                for entry in WalkDir::new(dirname) { ❸
                    println!("{}", entry?.path().display());
                }
            }
        }
    }
    Ok(())
}
```

❶ Use `fs::read_dir` to attempt reading a given directory.

❷ When this fails, print an error message and move on.

❸ Iterate over the directory entries and print their names.

Next, if the user has indicated only certain entry types, I should skip those entries that don't match:

```
pub fn run(config: Config) -> MyResult<()> {
    for dirname in config.dirs {
```

```
        match fs::read_dir(&dirname) {
            Err(e) => eprintln!("{}: {}", dirname, e),
            _ => {
                for entry in WalkDir::new(dirname) {
                    let entry = entry?; ❶
                    if let Some(types) = &config.entry_types { ❷
                        if !types.iter().any(|type_| match type_ {
❸
                            Link => entry.path_is_symlink(),
                            Dir => entry.file_type().is_dir(),
                            File => entry.file_type().is_file(),
                        }) {
                            continue; ❹
                        }
                    }
                    println!("{}", entry.path().display());
                }
            }
        }
    }
    Ok(())
}
```

❶ Unpack the `Result`.

❷ See if there are `Some(types)` to filter the entries.

❸ Use `Iterator::any` to see if any of the desired types matches the entry's type.

❹ Skip to the next entry when the condition is not met.

Recall that I used `Iterator::all` in Chapter 5 to return `true` if *all* of the elements in a vector passed some predicate. In the preceding code, I'm using `Iterator::any` to return `true` if *at least one* of the elements proves `true` for the predicate, which in this case is whether the entry's type matches one of the desired types. When I check the output, it seems to be finding, for instance, all the directories:

```
$ cargo run -- tests/inputs/ -t d
tests/inputs/
tests/inputs/a
tests/inputs/a/b
```

```
tests/inputs/a/b/c
tests/inputs/f
tests/inputs/d
tests/inputs/d/e
```

I can run **`cargo test type`** on Linux and macOS to verify that I'm now passing all of the tests that check for types alone. (Windows will fail because of the aforementioned lack of symbolic links.) The failures are for a combination of type and name, so next, I need to skip the filenames that don't match one of the given regular expressions:

```
pub fn run(config: Config) -> MyResult<()> {
    for dirname in config.dirs {
        match fs::read_dir(&dirname) {
            Err(e) => eprintln!("{}: {}", dirname, e),
            _ => {
                for entry in WalkDir::new(dirname) {
                    let entry = entry?;

                    // Same as before
                    if let Some(types) = &config.entry_types {}

                    if let Some(names) = &config.names {
                        if !names.iter().any(|re| {

re.is_match(&entry.file_name().to_string_lossy())
                        }) {
                            continue;
                        }
                    }

                    println!("{}", entry.path().display());
                }
            }
        }
    }
    Ok(())
}
```

I can use this to find, for instance, any regular file matching *mp3*, and it seems to work:

```
$ cargo run -- tests/inputs/ -t f -n mp3
```

```
tests/inputs/a/b/c/c.mp3
tests/inputs/d/e/e.mp3
```

If I run **`cargo test`** with this version of the program on a Unix-type platform, all tests pass. Huzzah! I could stop at this point, but I feel my code could be more elegant. I want to *refactor* this code, which means I want to restructure it without changing the way it works. Specifically, I don't like how I'm checking the types and names and using `continue` to skip entries. These are *filter* operations, so I'd like to use `Iterator::filter`. Following is my final `run` that still passes all the tests. Be sure you add `use walkdir::DirEntry` to your code for this:

```
pub fn run(config: Config) -> MyResult<()> {
    let type_filter = |entry: &DirEntry| match &config.entry_types
{ ❶
        Some(types) => types.iter().any(|t| match t {
            Link => entry.path_is_symlink(),
            Dir => entry.file_type().is_dir(),
            File => entry.file_type().is_file(),
        }),
        _ => true,
    };

    let name_filter = |entry: &DirEntry| match &config.names { ❷
        Some(names) => names
            .iter()
            .any(|re|
re.is_match(&entry.file_name().to_string_lossy())),
        _ => true,
    };

    for dirname in &config.dirs {
        match fs::read_dir(&dirname) {
            Err(e) => eprintln!("{}: {}", dirname, e),
            _ => {
                let entries = WalkDir::new(dirname)
                    .into_iter()
                    .filter_map(|e| e.ok()) ❸
                    .filter(type_filter) ❹
                    .filter(name_filter) ❺
                    .map(|entry|
entry.path().display().to_string()) ❻
                    .collect::<Vec<String>>(); ❼
```

```
                println!("{}", entries.join("\n"));
            }
        }
    }

    Ok(())
}
```

1. Create a closure to filter entries on `any` of the regular expressions.
2. Create a similar closure to filter entries by `any` of the types.
3. Turn `WalkDir` into an iterator and use `Iterator::filter_map` to select `Ok` values.
4. Filter out unwanted types.
5. Filter out unwanted names.
6. Turn each `DirEntry` into a string to display.
7. Use `Iterator::collect` to create a `Vec<String>`.

In the preceding code, I create two closures to use with `filter` operations. I chose to use closures because I wanted to capture values from the `config` as I first showed in Chapter 6. The first closure checks if any of the `config.entry_types` matches the `DirEntry::file_type`:

```
let type_filter = |entry: &DirEntry| match &config.entry_types {
    Some(types) => types.iter().any(|type_| match type_ { ❶
        Link => entry.path_is_symlink(), ❷
        Dir => entry.file_type().is_dir(), ❸
        File => entry.file_type().is_file(), ❹
    }),
    _ => true, ❹
};
```

1. Iterate over the `config.entry_types` to compare to the given entry. Note that `type` is a reserved word in Rust, so I use `type_`.
2. When the type is `Link`, return whether the entry is a symlink.
3. When the type is `Dir`, return whether the entry is a directory.

❹ When the type is `File`, return whether the entry is a file.

The preceding `match` takes advantage of the Rust compiler's ability to ensure that all variants of `EntryType` have been covered. For instance, comment out one arm like so:

```
let type_filter = |entry: &DirEntry| match &config.entry_types {
    Some(types) => types.iter().any(|t| match t {
        Link => entry.path_is_symlink(),
        Dir => entry.file_type().is_dir(),
        //File => entry.file_type().is_file(),
    }),
    _ => true,
};
```

The program will not compile, and the compiler warns that I've missed a case. You will not get this kind of safety if you use strings to model this. The `enum` type makes your code far safer and easier to verify and modify:

```
error[E0004]: non-exhaustive patterns: `&File` not covered
  --> src/lib.rs:95:51
   |
10 | / enum EntryType {
11 | |     Dir,
12 | |     File,
   | |     ---- not covered
13 | |     Link,
14 | | }
   | |_- `EntryType` defined here
...
95 |           Some(types) => types.iter().any(|t| match t {
   |                                                 ^ pattern
`&File`
   |                                                   not
covered
   |
   = help: ensure that all possible cases are being handled,
possibly by
     adding wildcards or more match arms
   = note: the matched value is of type `&EntryType`
```

The second closure is used to remove filenames that don't match one of the

given regular expressions:

```
let name_filter = |entry: &DirEntry| match &config.names {
    Some(names) => names ❶
        .iter()
        .any(|re|
re.is_match(&entry.file_name().to_string_lossy())),
    _ => true, ❷
};
```

❶ When there are `Some(names)`, use `Iterator::any` to check if the `DirEntry::file_name` matches any one of the regexes.

❷ When there are no regexes, return `true`.

The last piece I would like to highlight is the multiple operations I can chain together with iterators in the following code. As with reading lines from a file or entries in a directory, each value in the iterator is a `Result` that might yield a `DirEntry` value. I use `Iterator::filter_map` to map each `Result` into a closure that only allows values that yield an `Ok(DirEntry)` value. The `DirEntry` values are then passed to the two filters for types and names before being shunted to the `map` operation to transform them into `String` values.

```
let entries = WalkDir::new(dirname)
    .into_iter()
    .filter_map(|e| e.ok())
    .filter(type_filter)
    .filter(name_filter)
    .map(|entry| entry.path().display().to_string())
    .collect::<Vec<String>>();
```

While that is fairly compact code, I find it lean and expressive. I appreciate how much these functions are doing for me and how well they fit together. You are free to write code however you like so long as it passes the tests, but I find this to be my preferred solution.

Going Further

As with all the previous programs, I challenge you to implement all of the other features in `find`. For instance, two very useful options of `find` are `-max_depth` and `-min_depth` to control how deeply into the directory structure it should search. I notice there are `WalkDir::min_depth` and `WalkDir::max_depth` options you might use.

Next, perhaps try to find files by size. The `find` program has a particular syntax for indicating files less than, greater than, or exactly equal to sizes:

```
-size n[ckMGTP]
     True if the file's size, rounded up, in 512-byte blocks is n.
If
     n is followed by a c, then the primary is true if the file's
size
     is n bytes (characters).  Similarly if n is followed by a
scale
     indicator then the file's size is compared to n scaled as:

     k       kilobytes (1024 bytes)
     M       megabytes (1024 kilobytes)
     G       gigabytes (1024 megabytes)
     T       terabytes (1024 gigabytes)
     P       petabytes (1024 terabytes)
```

The `find` program can also take action on the results. For instance, there is a `-delete` option to remove an entry. This is useful for finding and removing empty files:

```
$ find . -size 0 -delete
```

I've often thought it would be nice to have a `-count` option to tell me how many items are found the way that `uniq -c` did in the last chapter. I can, of course, pipe this into `wc -l` (or, even better, `wcr`), but consider adding such an option to your program. Finally, I'd recommend you look at the source code for `fd`, another Rust replacement for `find`.

Summary

I hope you have an appreciation now for how complex real-world programs can become. The `find` program can combine multiple comparisons to help you find, say, the large files eating up your disk or files that haven't been modified in a long time which can be removed. Consider the skills you learned in this chapter:

- You can now use `Arg::possible_values` to constrain argument values to a limited set of strings, saving you time in validating user input.
- You can use `^` at the beginning of a regular expression to anchor the pattern to the beginning of the string and `$` at the end to anchor to the end of the string.
- You can create an `enum` type to represent alternate possibilities for a type. This provides far more security than using strings.
- You can use `WalkDir` to recursively search through a directory structure and evaluate the `DirEntry` values to find files, directories, and links.
- You learned how to chain multiple operations like `any`, `filter`, `map`, and `filter_map` with iterators.

---

1 This is one of those odd programs that has no short flags and the long flags start with a single dash.

2 Like Freud said, "Sometimes a dot is just a dot."

Chapter 8. Shave and a Haircut

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 8th Chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the author at kyclark@gmail.com.

I'm a mess since you cut me out but Chucky's arm keeps me company

—They Might Be Giants

The `cut` tool will excise text from a file or `STDIN`. The selected text could be some range of bytes or characters or might be fields denoted by a delimiter. In Chapter 4 (`headr`), you learned to select a contiguous range of characters or bytes, but this challenge goes further as the selections can be noncontiguous. Additionally, the output of the program can rearrange the selections, as in the string "3,1,5-7" which should select the third, first, and fifth through seventh bytes, characters, or fields. This will also be the first time you will deal with *delimited* text where some special character like a comma or a tab creates field boundaries. The challenge program will capture the spirit of the original tools but will not strive for complete fidelity as I will suggest a few changes which I feel are improvements.

What you will learn:

- How to read and write delimited text file using the `csv` module

- How to deference a value using `*`
- More on using `Iterator::filter_map` to combine `filter` and `map` operations

How cut Works

First let's review a portion for the BSD version of `cut`:

```
CUT(1)                    BSD General Commands Manual
CUT(1)

NAME
     cut -- cut out selected portions of each line of a file

SYNOPSIS
     cut -b list [-n] [file ...]
     cut -c list [file ...]
     cut -f list [-d delim] [-s] [file ...]

DESCRIPTION
     The cut utility cuts out selected portions of each line (as
specified by
     list) from each file and writes them to the standard output.
If no file
     arguments are specified, or a file argument is a single dash
('-'), cut
     reads from the standard input.  The items specified by list
can be in
     terms of column position or in terms of fields delimited by a
special
     character.  Column numbering starts from 1.

     The list option argument is a comma or whitespace separated
set of num-
     bers and/or number ranges.  Number ranges consist of a number,
a dash
     ('-'), and a second number and select the fields or columns
from the
     first number to the second, inclusive.  Numbers or number
ranges may be
     preceded by a dash, which selects all fields or columns from 1
to the
     last number.  Numbers or number ranges may be followed by a
```

```
dash, which
     selects all fields or columns from the last number to the end
of the
     line.  Numbers and number ranges may be repeated, overlapping,
and in any
     order.  If a field or column is specified multiple times, it
will appear
     only once in the output.  It is not an error to select fields
or columns
     not present in the input line.

     The options are as follows:

     -b list
             The list specifies byte positions.

     -c list
             The list specifies character positions.

     -d delim
             Use delim as the field delimiter character instead of
the tab
             character.

     -f list
             The list specifies fields, separated in the input by
the field
             delimiter character (see the -d option.)  Output
fields are sepa-
             rated by a single occurrence of the field delimiter
character.

     -n      Do not split multi-byte characters.  Characters will
only be out-
             put if at least one byte is selected, and, after a
prefix of zero
             or more unselected bytes, the rest of the bytes that
form the
             character are selected.

     -s      Suppress lines with no field delimiter characters.
Unless speci-
             fied, lines with no delimiters are passed through
unmodified.
```

As usual, the GNU version offers both short and long flags and several other

features:

```
NAME
       cut - remove sections from each line of files

SYNOPSIS
       cut OPTION... [FILE]...

DESCRIPTION
       Print selected parts of lines from each FILE to standard
output.

       Mandatory  arguments  to  long  options are mandatory for
short options
       too.

       -b, --bytes=LIST
              select only these bytes

       -c, --characters=LIST
              select only these characters

       -d, --delimiter=DELIM
              use DELIM instead of TAB for field delimiter

       -f, --fields=LIST
              select only these fields;  also print any line that
contains  no
              delimiter character, unless the -s option is
specified

       -n     with -b: don't split multibyte characters

       --complement
              complement the set of selected bytes, characters or
fields

       -s, --only-delimited
              do not print lines not containing delimiters

       --output-delimiter=STRING
              use  STRING  as  the  output delimiter the default is
to use the
              input delimiter

       --help display this help and exit
```

```
       --version
              output version information and exit

       Use one, and only one of -b, -c or -f.  Each LIST is  made
up  of  one
       range,  or  many ranges separated by commas.  Selected input
is written
       in the same order that it is read, and is written exactly
once.   Each
       range is one of:

       N      N'th byte, character or field, counted from 1

       N-     from N'th byte, character or field, to end of line

       N-M    from N'th to M'th (included) byte, character or field

       -M     from first to M'th (included) byte, character or
field

       With no FILE, or when FILE is -, read standard input.
```

Both tools implement the selection ranges in similar ways where numbers can be selected individually, in closed ranges like “1-3”, or in half-open ranges like “-3” to indicate 1 to 3 or “5-” to indicate 5 to the end. Additionally, the original tools will not allow a field to be repeated in the output and will rearrange them in ascending order. The challenge program will instead allow only for a comma-separated list of either single numbers or bounded ranges like “2-4” and will use the selections in the given order to create the output.

I’ll show you some examples of how the original tools work to the extent that the challenge program should implement. I will use the files found in the *08_cutr/tests/inputs* directory. First, consider a file with columns of information each in a fixed number of characters or so-called *fixed-width text*:

```
$ cd 08_cutr/tests/inputs
$ cat books.txt
Author              Year Title
Émile Zola          1865 La Confession de Claude
Samuel Beckett      1952 Waiting for Godot
Jules Verne         1870 20,000 Leagues Under the Sea
```

The *Author* column takes the first 20 characters:

```
$ cut -c 1-20 books.txt
Author
Émile Zola
Samuel Beckett
Jules Verne
```

The publication *Year* column occupies the next 5 characters:

```
$ cut -c 21-25 books.txt
Year
1865
1952
1870
```

The *Title* column occupies the last 30 characters. Note here that I intentionally request a larger range than exists to show that this is not considered an error:

```
$ cut -c 26-70 books.txt
Title
La Confession de Claude
Waiting for Godot
20,000 Leagues Under the Sea
```

I find it annoying that I cannot use this tool to rearrange the output such as by requesting the range for the *Title* followed by that for the *Author*:

```
$ cut -c 26-55,1-20 books.txt
Author              Title
Émile Zola          La Confession de Claude
Samuel Beckett      Waiting for Godot
Jules Verne         20,000 Leagues Under the Sea
```

I can grab just the first character like so:

```
$ cut -c 1 books.txt
A
É
S
```

```
J
```

As you've seen in previous chapters, bytes and characters are not always interchangeable. For instance, the "É" in "Émile Zola" is a Unicode character that is composed of two bytes, so asking for just one will result in an invalid character that is represented with the Unicode replacement character:

```
$ cut -b 1 books.txt
A
�
S
J
```

In my experience, fixed-width data files are less common than those where the columns of data are *delimited* with a character like a comma or a tab to show the boundaries of the data. Consider the same data in the file *books.tsv* where the file extension *.tsv* stands for *tab-separated values*:

```
$ cat books.tsv
Author  Year    Title
Émile Zola      1865    La Confession de Claude
Samuel Beckett  1952    Waiting for Godot
Jules Verne     1870    20,000 Leagues Under the Sea
```

By default, `cut` will assume the tab character is the field delimiter, so I can use the `-f` option to select, for instance, the publication year in the second column and the title in the third column like so:

```
$ cut -f 2,3 books.tsv
Year    Title
1865    La Confession de Claude
1952    Waiting for Godot
1870    20,000 Leagues Under the Sea
```

The comma is another common delimiter, and such files often have the extension *.csv* for *comma-separated values* (CSV). Following is the same data as a CSV file:

```
$ cat books.csv
```

```
Author,Year,Title
Émile Zola,1865,La Confession de Claude
Samuel Beckett,1952,Waiting for Godot
Jules Verne,1870,"20,000 Leagues Under the Sea"
```

To parse a CSV file, I must indicate the delimiter using the `-d` option. Note that I'm still unable to reorder the fields in the output as I indicate "2,1" for the second column followed by the first, but I get the columns back in their original order:

```
$ cut -d , -f 2,1 books.csv
Author,Year
Émile Zola,1865
Samuel Beckett,1952
Jules Verne,1870
```

You may have noticed that the third title contains a comma in *20,000* and so the title has been enclosed in quotes to indicate that the comma is not the field delimiter. This is a way to *escape* the delimiter or to tell the parser to ignore it. Unfortunately, both the BSD and GNU versions don't recognize this and will truncate the title prematurely:

```
$ cut -d , -f 1,3 books.csv
Author,Title
Émile Zola,La Confession de Claude
Samuel Beckett,Waiting for Godot
Jules Verne,"20
```

Noninteger values for any of the *list* option values that accept a list are rejected:

```
$ cut -f foo,bar books.tsv
cut: [-cf] list: illegal list value
```

Nonexistent files are handled in the course of processing, printing a message to STDERR that the file does not exist:

```
$ cut -c 1 books.txt blargh movies1.csv
A
```

```
É
S
J
cut: blargh: No such file or directory
t
T
L
```

Finally, the program will read `STDIN` by default or if the given input filename is the dash (`-`):

```
$ cat books.tsv | cut -f 2
Year
1865
1952
1870
```

The challenge program is expected to implement just this much of the original with the following changes:

1. Ranges must indicate both start and stop values (inclusive)
2. Output columns should be in the order specified by the user
3. Ranges may include repeated values
4. The parsing of delimited text files should respect escaped delimiters

Getting Started

The name of the challenge program should be `cutr` (pronounced *cutter*, I think) for a Rust version of `cut`. I recommend you begin with **`cargo new cutr`** and then copy the *08_cutr/tests* directory into your project. My solution will involve the following crates which you should add to your *Cargo.toml*:

```
[dependencies]
clap = "2.33"
csv = "1"
regex = "1"
```

```
[dev-dependencies]
assert_cmd = "1"
predicates = "1"
rand = "0.8"
```

Run **`cargo test`** to download the dependencies and run the tests, all of which should fail.

Defining the Arguments

I recommend the following structure for your *src/main.rs*:

```
fn main() {
    if let Err(e) = cutr::get_args().and_then(cutr::run) {
        eprintln!("{}", e);
        std::process::exit(1);
    }
}
```

Following is how I started my *src/lib.rs*. I want to highlight that I'm creating another `enum` as in Chapter 7, but this time the variants can hold a value. The value in this case will be another type alias I'm creating called `PositionList`, which is a `Vec<usize>`:

```
use crate::Extract::*; ❶
use clap::{App, Arg};
use std::error::Error;

type MyResult<T> = Result<T, Box<dyn Error>>;
type PositionList = Vec<usize>;  ❷

#[derive(Debug)] ❸
pub enum Extract {
    Fields(PositionList),
    Bytes(PositionList),
    Chars(PositionList),
}

#[derive(Debug)]
pub struct Config {
    files: Vec<String>, ❹
```

```
    delimiter: u8, ❺
    extract: Extract, ❻
}
```

❶ This allows me to use `Fields(...)` instead of `Extract::Fields(...)`.

❷ A `PositionList` is a vector of positive integer values.

❸ Define an `enum` to hold the variants for extracting fields, bytes, or characters.

❹ The `files` will be a vector of strings.

❺ The `delimiter` should be a single byte.

❻ The `extract` field will hold one of the `Extract` variants.

I decided to represent a range selection of something like “3,1,5-7” as `[3, 1, 5, 6, 7]`. Well, I actually subtract 1 from each value because I will be dealing with 0-offsets, but the point is that I thought it easiest to explicitly list all the positions in the order in which they will be selected. You may prefer to handle this differently.

You can start your `get_args` with the following:

```
pub fn get_args() -> MyResult<Config> {
    let matches = App::new("cutr")
        .version("0.1.0")
        .author("Ken Youens-Clark <kyclark@gmail.com>")
        .about("Rust cut")
        // What goes here?
        .get_matches();

    Ok(Config {
        files: ...
        delimiter: ...
        fields: ...
        bytes: ...
        chars: ...
    })
}
```

Begin your `run` by printing the `config`:

```
pub fn run(config: Config) -> MyResult<()> {
    println!("{:#?}", &config);
    Ok(())
}
```

See if you can get your program to print the following usage:

```
$ cargo run -- --help
cutr 0.1.0
Ken Youens-Clark <kyclark@gmail.com>
Rust cut

USAGE:
    cutr [OPTIONS] <FILE>...

FLAGS:
    -h, --help       Prints help information
    -V, --version    Prints version information

OPTIONS:
    -b, --bytes <BYTES>        Selected bytes
    -c, --chars <CHARS>        Selected characters
    -d, --delim <DELIMITER>    Field delimiter [default:        ]
    -f, --fields <FIELDS>      Selected fields

ARGS:
    <FILE>...    Input file(s) [default: -]
```

I wrote a function called `parse_pos` that works like the `parse_positive_int` function from Chapter 4. Here is how you might start it:

```
fn parse_pos(range: &str) -> MyResult<PositionList> { ❶
    unimplemented!();
}
```

❶ The function accepts a `&str` and might return a `PositionList`.

I have, of course, written a unit test for you. Add the following to your

src/lib.rs:

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

    #[test]
    fn test_parse_pos() {
        assert!(parse_pos("").is_err());

        let res = parse_pos("0");
        assert!(res.is_err());
        assert_eq!(res.unwrap_err().to_string(), "illegal list
value: \"0\"",);

        let res = parse_pos("a");
        assert!(res.is_err());
        assert_eq!(res.unwrap_err().to_string(), "illegal list
value: \"a\"",);

        let res = parse_pos("1,a");
        assert!(res.is_err());
        assert_eq!(res.unwrap_err().to_string(), "illegal list
value: \"a\"",);

        let res = parse_pos("2-1");
        assert!(res.is_err());
        assert_eq!(
            res.unwrap_err().to_string(),
            "First number in range (2) must be lower than second
number (1)"
        );

        let res = parse_pos("1");
        assert!(res.is_ok());
        assert_eq!(res.unwrap(), vec![0]);

        let res = parse_pos("1,3");
        assert!(res.is_ok());
        assert_eq!(res.unwrap(), vec![0, 2]);

        let res = parse_pos("1-3");
        assert!(res.is_ok());
        assert_eq!(res.unwrap(), vec![0, 1, 2]);

        let res = parse_pos("1,7,3-5");
        assert!(res.is_ok());
```

```
        assert_eq!(res.unwrap(), vec![0, 6, 2, 3, 4]);
    }
}
```

At this point, I expect you can read the above code well enough to understand exactly how the function should work. I recommend you stop reading at this point and write the code that will pass this test.

After **`cargo test test_parse_pos`** passes, your program should reject an invalid range and print an error message:

```
$ cargo run -- -f foo,bar tests/inputs/books.tsv
illegal list value: "foo"
```

It should also reject invalid ranges:

```
$ cargo run -- -f 3-2 tests/inputs/books.tsv
First number in range (3) must be lower than second number (2)
```

When given valid arguments, your program should be able to display a structure like so:

```
$ cargo run -- -f 1 -d , tests/inputs/movies1.csv
Config {
    files: [
        "tests/inputs/movies1.csv", ❶
    ],
    delimiter: 44, ❷
    extract: Fields( ❸
        [
            0,
        ],
    ),
}
```

❶ The positional argument goes into `files`.

❷ The `-d` value of a comma has a byte value of `44`.

❸ The `-f` value of 1 creates the `Extract::Fields([0])` variant.

When parsing a TSV file, use the tab as the default delimiter, which has a byte value of `9`:

```
$ cargo run -- -f 2-3 tests/inputs/movies1.tsv
Config {
    files: [
        "tests/inputs/movies1.tsv",
    ],
    delimiter: 9,
    extract: Fields(
        [
            1,
            2,
        ],
    ),
}
```

Note that the options for `-f|--fields`, `-b|--bytes`, and `-c|--chars` are all mutually exclusive and should be rejected:

```
$ cargo run -- -f 1 -b 8-9 tests/inputs/movies1.tsv
error: The argument '--fields <FIELDS>' cannot be used with '--
bytes <BYTES>'

USAGE:
    cutr <FILE>... --bytes <BYTES> --delim <DELIMITER> --fields
<FIELDS>
```

Try to get just this much of your program working before you proceed. You should be able to pass **`cargo test dies`**:

```
running 8 tests
test dies_chars_bytes ... ok
test dies_chars_bytes_fields ... ok
test dies_chars_fields ... ok
test dies_bytes_fields ... ok
test dies_not_enough_args ... ok
test dies_bad_digit_field ... ok
test dies_bad_digit_bytes ... ok
test dies_bad_digit_chars ... ok
```

Parsing the Position List

I assume you wrote a passing `parse_pos` function, so compare your version to mine:

```
fn parse_pos(range: &str) -> MyResult<PositionList> {
    let mut fields: Vec<usize> = vec![]; ❶
    let range_re = Regex::new(r"(\d+)?-(\d+)?").unwrap(); ❷
    for val in range.split(',') { ❸
        if let Some(cap) = range_re.captures(val) { ❹
            let n1: &usize = &cap[1].parse()?; ❺
            let n2: &usize = &cap[2].parse()?;

            if n1 < n2 { ❻
                for n in *n1..=*n2 { ❼
                    fields.push(n);
                }
            } else {
                return Err(From::from(format!( ❽
                    "First number in range ({}) \
                    must be lower than second number ({})",
                    n1, n2
                )));
            }
        } else {
            match val.parse() { ❾
                Ok(n) if n > 0 => fields.push(n),
                _ => {
                    return Err(From::from(format!(
                        "illegal list value: \"{}\"",
                        val
                    )))
                }
            }
        }
    }

    // Subtract one for field indexes
    Ok(fields.into_iter().map(|i| i - 1).collect()) ❿
}
```

❶ Create a mutable vector to hold all the positions.

❷ Create a regular expression to capture two numbers separated by a dash.

❸ Split the range values on a comma.

❹ See if this part of the list matches the regex.

❺ Convert the two captured numbers to `usize` integer values.

❻ If the first number is less than the second, iterate through the range and add the values to `fields`.

❼ Use the `*` operator to dereference the two number values.

❽ Return an error about an invalid range.

❾ If it's possible to convert the value to a `usize` integer, add it to the list or else throw an error.

❿ Return the given list with all the values adjusted down by 1.

> **NOTE**
>
> In the regular expression, I use `r""` to denote a *raw* string so that Rust won't try to interpret the string. For instance, you've seen that Rust will interpret `\n` as a newline. Without this, the compiler would complain that `\d` is an *unknown character escape*:

```
error: unknown character escape: `d`
   --> src/lib.rs:155:34
    |
155 |     let range_re = Regex::new("(\d+)?-(\d+)?").unwrap();
    |                                  ^ unknown character escape
    |
    = help: for more information, visit <https://static.rust-
lang.org
      /doc/master/reference.html#literals>
```

I would like to highlight two new pieces of syntax in the preceding code. First, I used parentheses in the regular expression `(\d+)-(\d+)` to indicate one or more digits followed by a dash followed by one or more digits as shown in Figure 8-1. If the regular expression matches the given string, then I can use the `Regex::captures` to extract the digits from the string. Note that they are available in 1-based counting, so the contents of the first capturing parentheses are available in position `1` of the captures. Because the captured values matched digit characters, they should be parsable as `usize` values.

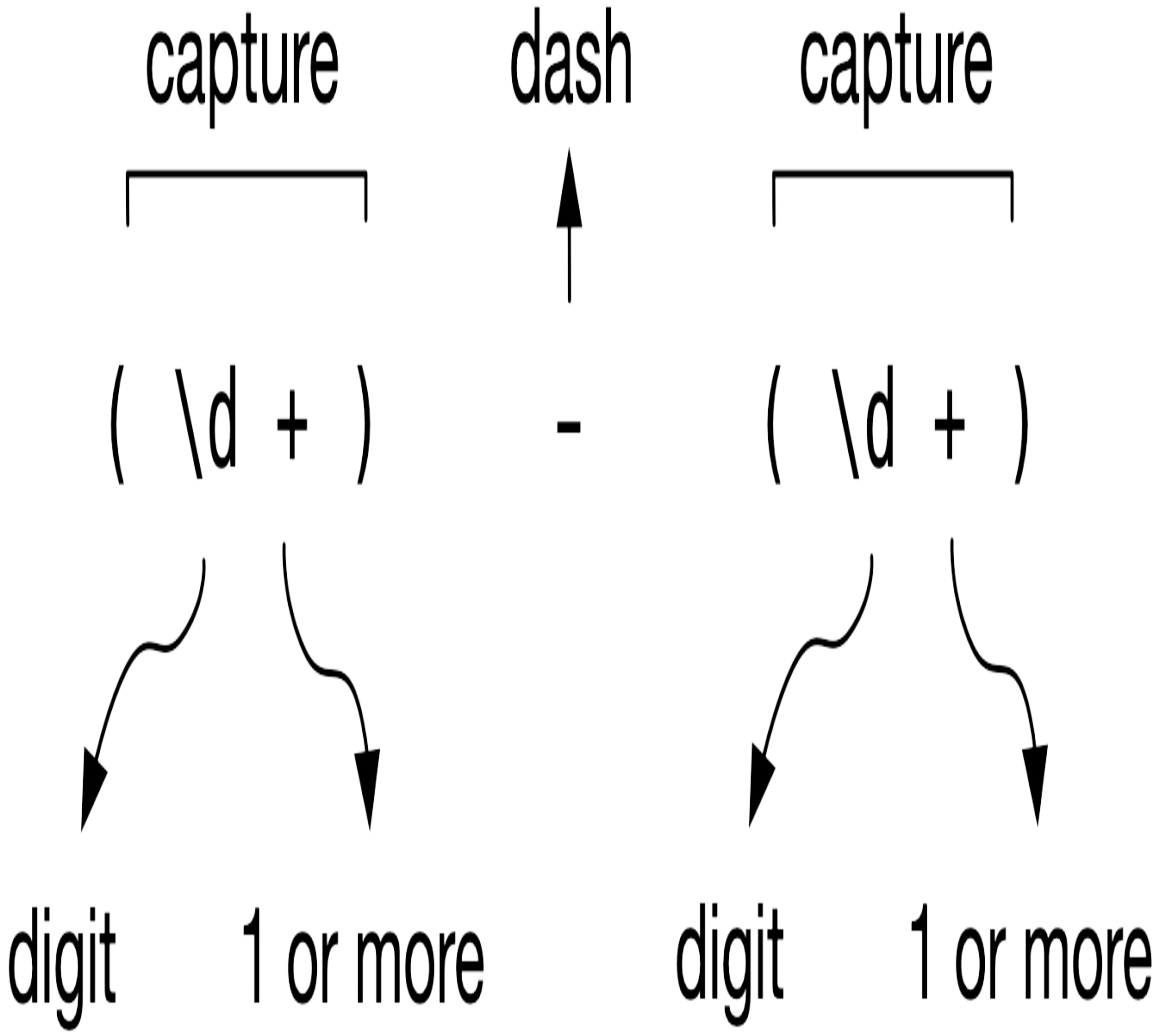


Figure 8-1. The parentheses in the regular expression will capture the values they surround

The second piece of syntax is the `*` operator in `for n in *n1..=*n2`. If you remove these and try to compile the code, you will see the following error:

```
error[E0277]: the trait bound `&usize: Step` is not satisfied
   --> src/lib.rs:165:34
    |
165 |                         for n in n1..=n2 {
    |                                  ^^^^^^^ the trait `Step` is
not
    |                                          implemented for
`&usize`
```

This is one case where the compiler's message is a bit cryptic and does not include the solution. The problem is that `n1` and `n2` are `&usize` references. A reference is a pointer to a piece of memory, not a copy of the value, and so the pointer must be *dereferenced* to use the underlying value. There are many times when Rust silently dereferences values, but this is one time when the `*` operator is required.

Here is how I incorporate these ideas into my `get_args`. First, I define all the arguments:

```
pub fn get_args() -> MyResult<Config> {
    let matches = App::new("cutr")
        .version("0.1.0")
        .author("Ken Youens-Clark <kyclark@gmail.com>")
        .about("Rust cut")
        .arg(
            Arg::with_name("files") ❶
                .value_name("FILE")
                .help("Input file(s)")
                .required(true)
                .default_value("-")
                .min_values(1),
        )
        .arg(
            Arg::with_name("delimiter") ❷
                .value_name("DELIMITER")
                .help("Field delimiter")
                .short("d")
                .long("delim")
                .default_value("\t"),
        )
        .arg(
            Arg::with_name("fields") ❸
                .value_name("FIELDS")
                .help("Selected fields")
                .short("f")
                .long("fields")
                .conflicts_with_all(&["chars", "bytes"]),
        )
        .arg(
            Arg::with_name("bytes") ❹
                .value_name("BYTES")
                .help("Selected bytes")
                .short("b")
```

```
                .long("bytes")
                .conflicts_with_all(&["fields", "chars"]),
        )
        .arg(
            Arg::with_name("chars") ❺
                .value_name("CHARS")
                .help("Selected characters")
                .short("c")
                .long("chars")
                .conflicts_with_all(&["fields", "bytes"]),
        )
        .get_matches();
```

❶ The required `files` accepts multiple values and defaults to the dash.

❷ The `delimiter` uses the tab as the default value.

❸ The `fields` option conflicts with `chars` and `bytes`.

❹ The `bytes` option conflicts with `fields` and `chars`.

❺ The `chars` options conflicts with `fields` and `bytes`.

Next, I convert the delimiter to bytes and verify that there is only one:

```
    let delimiter = matches.value_of("delimiter").unwrap_or("\t");
    let delim_bytes = delimiter.as_bytes();
    if delim_bytes.len() > 1 {
        return Err(From::from(format!(
            "--delim \"{}\" must be a single byte",
            delimiter
        )));
    }
```

I use the `parse_pos` function to handle all the optional list values:

```
    let fields =
matches.value_of("fields").map(parse_pos).transpose()?;
    let bytes =
matches.value_of("bytes").map(parse_pos).transpose()?;
    let chars =
matches.value_of("chars").map(parse_pos).transpose()?;
```

> **NOTE**
>
> I'm introducing `Option::transpose` here that "transposes an `Option` of a `Result` into a `Result` of an `Option`."

I then figure out which `Extract` variant to create. I should never trigger the `else` clause in this code, but it's good to have:

```
        let extract = if let Some(field_pos) = fields {
            Fields(field_pos)
        } else if let Some(byte_pos) = bytes {
            Bytes(byte_pos)
        } else if let Some(char_pos) = chars {
            Chars(char_pos)
        } else {
            return Err(From::from("Must have --fields, --bytes, or --
chars"));
        };
```

If the code makes it to this point, then I appear to have valid arguments that I can return:

```
        Ok(Config {
            files: matches.values_of_lossy("files").unwrap(),
            delimiter: delim_bytes[0],
            extract,
        })
}
```

Next, you will need to figure out how you will use this information to extract the desired bits from the inputs.

Extracting Characters or Bytes

In Chapters 4 (`headr`) and 5 (`wcr`), you learned how to process lines, bytes, and characters in a file. You should draw on those programs to help you select characters and bytes in this challenge. One difference is that line endings need not be preserved, so you may use `BufRead::lines` to read

the lines of input text.

To start, you might consider bringing in the `open` function used before to help open each file:

```
fn open(filename: &str) -> MyResult<Box<dyn BufRead>> {
    match filename {
        "-" => Ok(Box::new(BufReader::new(io::stdin()))),
        _ => Ok(Box::new(BufReader::new(File::open(filename)?))),
    }
}
```

You can expand your `run` to handle good and bad files:

```
pub fn run(config: Config) -> MyResult<()> {
    for filename in &config.files {
        match open(filename) {
            Err(err) => eprintln!("{}: {}", filename, err),
            Ok(_file) => println!("Opened {}", filename),
        }
    }
    Ok(())
}
```

This will require some more imports:

```
use crate::Extract::*;
use clap::{App, Arg};
use regex::Regex;
use std::{
    error::Error,
    fs::File,
    io::{self, BufRead, BufReader},
};
```

Now consider how you might extract characters from each line of the filehandle. I wrote a function called `extract_chars` that will return a new string composed of the characters at the given index positions:

```
fn extract_chars(line: &str, char_pos: &[usize]) -> String {
    unimplemented!();
}
```

TIP

I'm using `&[usize]` instead of `&PositionList` because of the suggestion from **`cargo clippy`**:

```
warning: writing `&Vec<_>` instead of `&[_]` involves one more
reference
and cannot be used with non-Vec-based slices
   --> src/lib.rs:214:40
    |
214 | fn extract_chars(line: &str, char_pos: &PositionList) ->
String {
    |                                        ^^^^^^^^^^^^^
    |
    = note: `#[warn(clippy::ptr_arg)]` on by default
    = help: for further information visit
    https://rust-lang.github.io/rust-
clippy/master/index.html#ptr_arg
```

Here is a test you can add to your `tests` module to help you see how the function might work:

```
#[test]
fn test_extract_chars() {
    assert_eq!(extract_chars("", &[0]), "".to_string());
    assert_eq!(extract_chars("ábc", &[0]), "á".to_string());
    assert_eq!(extract_chars("ábc", &[0, 2]), "ác".to_string());
    assert_eq!(extract_chars("ábc", &[0, 1, 2]),
"ábc".to_string());
    assert_eq!(extract_chars("ábc", &[2, 1]), "cb".to_string());
    assert_eq!(extract_chars("ábc", &[0, 1, 4]), "áb".to_string());
}
```

I also wrote a similar `extract_bytes` function to parse out bytes:

```
fn extract_bytes(line: &str, byte_pos: &[usize]) -> String {
    unimplemented!();
)
```

Here is the test:

```
fn test_extract_bytes() {
    assert_eq!(extract_bytes("ábc", &[0]), "�".to_string());
    assert_eq!(extract_bytes("ábc", &[0, 1]), "á".to_string());
    assert_eq!(extract_bytes("ábc", &[0, 1, 2]), "áb".to_string());
    assert_eq!(extract_bytes("ábc", &[0, 1, 2, 3]),
"ábc".to_string());
    assert_eq!(extract_bytes("ábc", &[3, 2]), "cb".to_string());
    assert_eq!(extract_bytes("ábc", &[0, 1, 5]), "á".to_string());
}
```

Once you have written these two functions so that they pass the given test, iterate the lines of input text and use the preceding functions to extract and print the desired characters or bytes from each line. You should be able to pass all but the following when you run **`cargo test`**:

```
failures:
    csv_f1
    csv_f1_2
    csv_f1_3
    csv_f2
    csv_f2_3
    csv_f3
    tsv_f1
    tsv_f1_2
    tsv_f1_3
    tsv_f2
    tsv_f2_3
    tsv_f3
```

Parsing Delimited Text Files

To pass the final tests, you will need to learn how to parse delimited text files. As stated earlier, this file format uses some delimiter like a comma or tab to indicate the boundaries of a field. Sometimes the delimiting character may also be part of the data, in which case the field should be enclosed in quotes to escape the delimiter. The easiest way to properly parse delimited text is to use something like the `csv` module. I highly recommend that you first read the tutorial.

Next, consider the following example that shows how you can use this module to parse delimited data. If you would like to compile and run this

code, start a new project and use the following for *src/main.rs*. Be sure to add the `csv` dependency to your *Cargo.toml* and copy the *books.csv* file into the project.

```
use csv::StringRecord;
use std::fs::File;

fn main() -> std::io::Result<()> {
    let mut reader = csv::ReaderBuilder::new() ❶
        .delimiter(b',') ❷
        .from_reader(File::open("books.csv")?); ❸

    fmt(reader.headers()?); ❹
    for record in reader.records() { ❺
        fmt(&record?);
    }

    Ok(())
}

fn fmt(rec: &StringRecord) {
    println!(
        "{}",
        rec.into_iter() ❻
            .map(|v| format!("{:20}", v)) ❼
            .collect::<Vec<String>>() ❽
            .join("")
    )
}
```

❶ Use `csv::ReaderBuilder` to parse a file.

❷ The `delimiter` must be a single `u8` byte.

❸ The `from_reader` method accepts a value that implements the `Read` trait.

❹ The `Reader::headers` will return the column names in the first row as a `StringRecord`.

❺ The `Reader::records` method provides access to an iterator over `StringRecord` values.

❻ The field values in a `StringRecord` can be iterated.

❼ Use `Iterator::map` to format the values into a field 20 characters wide.

❽ Collect the strings into a vector and join them into a new string for printing.

If I run this program, I will see the following output:

```
$ cargo run
Author              Year                Title
Émile Zola          1865                La Confession de Claude
Samuel Beckett      1952                Waiting for Godot
Jules Verne         1870                20,000 Leagues Under the
Sea
```

Coming back to the challenge program, think about how you might use some of these ideas to write a function like `extract_fields` that accepts a `StringRecord` and pulls out the fields found in the `PositionList`:

```
fn extract_fields(record: &StringRecord, field_pos: &[usize]) ->
Vec<String> {
    unimplemented!();
}
```

Here is a test you could use:

```
#[test]
fn test_extract_fields() {
    let rec = StringRecord::from(vec!["Captain", "Sham", "12345"]);
    assert_eq!(extract_fields(&rec, &[0]), &["Captain"]);
    assert_eq!(extract_fields(&rec, &[1]), &["Sham"]);
    assert_eq!(extract_fields(&rec, &[0, 2]), &["Captain",
"12345"]);
    assert_eq!(extract_fields(&rec, &[0, 3]), &["Captain"]);
    assert_eq!(extract_fields(&rec, &[1, 0]), &["Sham",
"Captain"]);
}
```

I think that might be enough to help you find a solution. This is a challenging program, so don’t give up too quickly. Fear is the mind-killer.

Solution

I'll show you my solution, starting with how I select the characters:

```
fn extract_chars(line: &str, char_pos: &[usize]) -> String {
    let chars: Vec<_> = line.chars().collect(); ❶
    char_pos.iter().filter_map(|i| chars.get(*i)).collect() ❷
}
```

❶ Break the line into a vector of characters. The `Vec` type annotation is required by Rust because `Iterator::collect` can return many different types of collections.

❷ Use `Iterator::filter_map` with `Vec::get` to select valid character positions and collect them into a new string.

The `filter_map` function, as you might imagine, combines the operations of `filter` and `map`. The closure uses `chars.get(*i)` in an attempt to select the character at the given index. This might fail if the user has requested positions beyond the end of the string, but a failure to select a character should not generate an error. `Vec::get` will return an `Option<char>`, and `filter_map` will skip all the `None` values and unwrap the `Some<char>` values. Here is a longer way to write this:

```
fn extract_chars(line: &str, char_pos: &[usize]) -> String {
    let chars: Vec<char> = line.chars().collect();
    char_pos
        .iter()
        .map(|i| chars.get(*i)) ❶
        .filter(|v| v.is_some()) ❷
        .map(|v| v.unwrap()) ❸
        .collect::<String>() ❹
}
```

❶ Try to get the characters as the given index positions.

❷ Filter out the `None` values.

❸ Unwrap the `Some` values.

❹

Collect the filtered characters into a `String`.

In the preceding code, I use `*i` to dereference the value similar to earlier in the chapter. If I remove the `*`, the compiler would complain thusly:

```
error[E0277]: the type `[char]` cannot be indexed by `&usize`
   --> src/lib.rs:227:35
    |
227 |         .filter_map(|i| chars.get(i))
    |                                   ^ slice indices are of type
    |                                     `usize` or ranges of
`usize`
    |
    = help: the trait `SliceIndex<[char]>` is not implemented for
`&usize`
```

The error message is vague on how to fix this, but the problem is that `i` is a `&usize` but I need a type `usize`. The deference `*` removes the `&` reference, hence the name.

The selection of bytes is very similar, but I have to deal with the fact that bytes must be explicitly cloned. As with `extract_chars`, the goal is to return a new string, but there is a potential problem if the byte selection breaks Unicode characters and so produces an invalid UTF-8 string:

```
fn extract_bytes(line: &str, byte_pos: &[usize]) -> String {
    let bytes = line.as_bytes(); ❶
    let selected: Vec<u8> = byte_pos
        .iter()
        .filter_map(|i| bytes.get(*i)) ❷
        .cloned() ❸
        .collect();
    String::from_utf8_lossy(&selected).into_owned() ❹
}
```

❶ Break the line into a vector of bytes.

❷ Use `filter_map` to select bytes at the wanted positions.

❸ Clone the resulting `Vec<&u8>` into a `Vec<u8>` to remove the references.

❹ Use `String::from_utf8_lossy` to generate a string from possibly invalid bytes.

You may wonder why I used `Iterator::clone` in the preceding code. Let me show you the error message if I remove it:

```
error[E0277]: a value of type `Vec<u8>` cannot be built from
an iterator over elements of type `&u8`
   --> src/lib.rs:215:10
    |
215 |         .collect();
    |          ^^^^^^^ value of type `Vec<u8>` cannot be built from
    |                  `std::iter::Iterator<Item=&u8>`
    |
    = help: the trait `FromIterator<&u8>` is not implemented for
`Vec<u8>`
```

The `filter_map` will produce a `Vec<&u8>`, which is a vector of references to `u8` values, but `String::from_utf8_lossy` expects `&[u8]`, a slice of bytes. As the `Iterator::clone` documentation notes, this method "Creates an iterator which clones all of its elements. This is useful when you have an iterator over `&T`, but you need an iterator over `T`."

Finally, here is one way to extract the fields from a `csv::StringRecord`:

```
fn extract_fields(record: &StringRecord, field_pos: &[usize]) ->
Vec<String> {
    field_pos
        .iter()
        .filter_map(|i| record.get(*i)) ❶
        .map(|v| v.to_string()) ❷
        .collect() ❸
}
```

❶ Use `csv::StringRecord::get` to try to get the field for the index position.

❷ Use `Iterator::map` to turn `&str` values into `String` values.

❸ Collect the results into a `Vec<String>`.

I would like to show you another way to write this function so that it will return a `Vec<&str>` which will be slightly more memory efficient as it will not have to make copies of the strings. The tradeoff is that I must indicate the lifetimes. First, let me naïvely try to write it like so:

```
// This will not compile
fn extract_fields(record: &StringRecord, field_pos: &[usize]) ->
Vec<&str> {
    field_pos.iter().filter_map(|i| record.get(*i)).collect()
}
```

If I try to compile this, the Rust compiler will complain about lifetimes and will suggest the following changes:

```
help: consider introducing a named lifetime parameter
    |
162 | fn extract_fields<'a>(record: &'a StringRecord, field_pos:
&'a [usize])
-> Vec<&'a str> {
```

I will change the function definition to the proposed version:

```
fn extract_fields<'a>( ❶
    record: &'a StringRecord,
    field_pos: &'a PositionList,
) -> Vec<&'a str> {
    field_pos.iter().filter_map(|i| record.get(*i)).collect() ❷
}
```

❶ Indicate the same lifetime `'a` for all the values.

❷ I have removed the step to convert each value to a `String`.

Both versions will pass the unit test. The latter version is slightly more efficient and shorter but also has more cognitive overhead for the reader. Choose whichever version you feel you'll be able to understand six weeks from now.

Here is my final `run` function that incorporates all these ideas and will pass all the tests:

```
pub fn run(config: Config) -> MyResult<()> {
    for filename in &config.files {
        match open(filename) {
            Err(err) => eprintln!("{}: {}", filename, err),
            Ok(file) => match &config.extract {
                Fields(field_pos) => {
                    let mut reader = ReaderBuilder::new() ❶
                        .delimiter(config.delimiter)
                        .has_headers(false)
                        .from_reader(file);

                    let mut wtr = WriterBuilder::new() ❷
                        .delimiter(config.delimiter)
                        .from_writer(io::stdout());

                    for record in reader.records() { ❸
                        let record = record?;
                        wtr.write_record(extract_fields( ❹
                            &record, field_pos,
                        ))?;
                    }
                }
                Bytes(byte_pos) => {
                    for line in file.lines() { ❺
                        println!("{}", extract_bytes(line?,
&byte_pos));
                    }
                }
                Chars(char_pos) => {
                    for line in file.lines() { ❻
                        println!("{}", extract_chars(line?,
&char_pos));
                    }
                }
            },
        }
    }
    Ok(())
}
```

❶ If the user has requested fields from a delimited file, use `csv::ReaderBuilder` to create a mutable reader using the given delimiter. Do not attempt to parse a header row.

❷ Use `csv::WriterBuilder` to write the output to `STDOUT` using the input delimiter.

❸ Iterate through the records.

❹ Write the extracted fields to the output.

❺ Iterate the lines of text and print the extract bytes.

❻ Iterate the lines of text and print the extract characters.

> **NOTE**
>
> I use `csv::WriterBuilder` to correctly escape enclosed delimiters in fields. None of the tests require this, so you may have found a simpler way to write the output that passes the tests. You will shortly see why I did this.

In the preceding code, you may be curious why I ignore any possible headers in the delimited files. By default, the `csv::Reader` will attempt to parse the first row for the column names, but I don't need to do anything special with these values in this program. If I used this default behavior, I would have to handle the headers separately from the rest of the records. In this context, it's easier to treat the first row like any other record.

This program passes all the tests and seems to work pretty well for all the testing input files. Because I'm using the `csv` module to parse delimited text files and write the output, this program will correctly handle delimited text files, unlike the original `cut` programs. I'll use *tests/inputs/books.csv* again to demonstrate that `cutr` will correctly select a field containing the delimiter and will create output that properly escapes the delimiter:

```
$ cargo run -- -d , -f 1,3 tests/inputs/books.csv
Author,Title
Émile Zola,La Confession de Claude
Samuel Beckett,Waiting for Godot
Jules Verne,"20,000 Leagues Under the Sea"
```

These choices make `cutr` unsuitable as a direct replacement for `cut` as many uses may count on the behavior of the original tool. As Ralph Waldo Emerson said, "A foolish consistency is the hobgoblin of little minds." I don't believe all these tools need to mimic the original tools, especially when

this seems to be such an improvement.

Going Further

- Make `cutr` parse delimited files exactly like the original tools and have the "correct" parsing of delimited files be an option.
- Implement the partial ranges like *-3* to mean *1-3* or *5-* to mean *5 to the end*. Be aware that trying to run **`cargo run -- -f -3 tests/inputs/books.tsv`** will cause `clap` to interpret `-3` as an option. Use `-f=-3` instead.
- Currently the `--delimiter` for parsing input delimited files is also used for the output delimiter. Add an option to change this but have it default to the input delimiter.
- Add an output filename option that defaults to `STDOUT`.
- Check out the `xsv`, a "fast CSV command line toolkit written in Rust."

Summary

Lift your gaze upon the knowledge you gained in this exercise:

- You've learned how to dereference a value using the `*`. Sometimes the compiler messages indicate that this is the solution, but other times you must infer this syntax when, for instance, you have `&usize` but need `usize`. Remember that the `*` essentially removes the `&`.
- The `Iterator::filter_map` combines `filter` and `map` for more concise code. You used this with a `get` idea that works both for selecting positions from a vector or fields from a `StringRecord`, which might fail and so are removed from the results.

- You compared how to return a `String` versus a `&str` from a function, the latter of which required indicating lifetimes.
- You can now parse and create delimited text using the `csv` module. While we only looked at files delimited by commas and tab characters, there are many other delimiters in the wild. CSV files are some of the most common data formats, so these patterns will likely prove very useful.

Chapter 9. Jack the Grepper

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 9th Chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the author at kyclark@gmail.com.

Please explain the expression on your face

—They Might Be Giants

The `grep` program[1] will find lines of input that match a given regular expression. By default, the input can come from `STDIN`, but you can also provide the names of one or more files or directories if you also use a recursive option to find all the files in those directories. The normal output will be the lines that match the given pattern, but you can invert the match to find the lines that don't match. You can also instruct `grep` to print the number of matching lines instead of the lines. Pattern matching is normally case-sensitive, but you can use an option to perform case-insensitive matching. While the original program will do more, the challenge program will only go this far.

You will learn:

- How to use a case-sensitive regular expression

- About variations of regular expression syntax
- Another syntax to indicate a trait bound
- How to use Rust's logical *AND* (&&) and *OR* (||) operators

How grep Works

The manual page for the BSD `grep` shows just how many different options the command will accept:

```
GREP(1)                   BSD General Commands Manual
GREP(1)

NAME
     grep, egrep, fgrep, zgrep, zegrep, zfgrep -- file pattern
searcher

SYNOPSIS
     grep [-abcdDEFGHhIiJLlmnOopqRSsUVvwxZ] [-A num] [-B num] [-
C[num]]
          [-e pattern] [-f file] [--binary-files=value] [--
color[=when]]
          [--colour[=when]] [--context[=num]] [--label] [--line-
buffered]
          [--null] [pattern] [file ...]

DESCRIPTION
     The grep utility searches any given input files, selecting
lines that
     match one or more patterns.  By default, a pattern matches an
input line
     if the regular expression (RE) in the pattern matches the
input line
     without its trailing newline.  An empty expression matches
every line.
     Each input line that matches at least one of the patterns is
written to
     the standard output.

     grep is used for simple patterns and basic regular expressions
(BREs);
     egrep can handle extended regular expressions (EREs).  See
re_format(7)
```

```
     for more information on regular expressions.  fgrep is quicker
than both
     grep and egrep, but can only handle fixed patterns (i.e. it
does not
     interpret regular expressions).  Patterns may consist of one
or more
     lines, allowing any of the pattern lines to match a portion of
the input.
```

The GNU version is very similar:

```
GREP(1)                     General Commands Manual
GREP(1)

NAME
       grep, egrep, fgrep - print lines matching a pattern

SYNOPSIS
       grep [OPTIONS] PATTERN [FILE...]
       grep [OPTIONS] [-e PATTERN | -f FILE] [FILE...]

DESCRIPTION
       grep  searches the named input FILEs (or standard input if
no files are
       named, or if a single hyphen-minus (-) is given as file
name) for lines
       containing  a  match to the given PATTERN.  By default, grep
prints the
       matching lines.
```

To demonstrate how `grep` works, I'll use the *09_grepr/tests/inputs* directory:

```
$ cd 09_grepr/tests/inputs
$ wc -l *
       9 bustle.txt
       0 empty.txt
       1 fox.txt
       9 nobody.txt
      19 total
```

To start verify for yourself that **`grep fox empty.txt`** will print nothing when using the *empty.txt* file. As shown by the usage, `grep` accepts a regular

expression as the first positional argument and possibly some input files for the rest. Note that an empty regular expression will match all lines of input, and here I'll use the input file *fox.txt*, which contains one line of text:

```
$ grep "" fox.txt
The quick brown fox jumps over the lazy dog.
```

Take a peek at this lovely Emily Dickinson poem, and notice that "Nobody" is always capitalized:

```
$ cat nobody.txt
I'm Nobody! Who are you?
Are you—Nobody—too?
Then there's a pair of us!
Don't tell! they'd advertise—you know!

How dreary—to be—Somebody!
How public—like a Frog—
To tell one's name—the livelong June—
To an admiring Bog!
```

If I search for *Nobody*, the two lines containing the string are printed:

```
$ grep Nobody nobody.txt
I'm Nobody! Who are you?
Are you—Nobody—too?
```

If I search for lowercase "nobody" with **`grep nobody nobody.txt`**, nothing is printed. I can, however, use `-i|--ignore-case` to find these lines:

```
$ grep -i nobody nobody.txt
I'm Nobody! Who are you?
Are you—Nobody—too?
```

I can use `-v|--invert-match` option to find the lines that don't match the pattern:

```
$ grep -v Nobody nobody.txt
Then there's a pair of us!
```

```
Don't tell! they'd advertise—you know!

How dreary—to be—Somebody!
How public—like a Frog—
To tell one's name—the livelong June—
To an admiring Bog!
```

The `-c|--count` option will cause the output to be a summary of the number of times a match occurs:

```
$ grep -c Nobody nobody.txt
2
```

I can combine `-v` and `-c` to count the lines not matching:

```
$ grep -vc Nobody nobody.txt
7
```

When searching multiple input files, the output includes filename:

```
$ grep The *.txt
bustle.txt:The bustle in a house
bustle.txt:The morning after death
bustle.txt:The sweeping up the heart,
fox.txt:The quick brown fox jumps over the lazy dog.
nobody.txt:Then there's a pair of us!
```

The filename is also included for the counts:

```
$ grep -c The *.txt
bustle.txt:3
empty.txt:0
fox.txt:1
nobody.txt:1
```

Normally, the positional arguments are files, and the inclusion of a directory such as my *$HOME* directory will cause `grep` to print a warning:

```
$ grep The bustle.txt $HOME fox.txt
bustle.txt:The bustle in a house
bustle.txt:The morning after death
```

```
bustle.txt:The sweeping up the heart,
grep: /Users/kyclark: Is a directory
fox.txt:The quick brown fox jumps over the lazy dog.
```

Directory names are only acceptable when using the `-r|--recursive` option to find all the files in a directory that contain matching text:

```
$ grep -r The .
./nobody.txt:Then there's a pair of us!
./bustle.txt:The bustle in a house
./bustle.txt:The morning after death
./bustle.txt:The sweeping up the heart,
./fox.txt:The quick brown fox jumps over the lazy dog.
```

The `-r` and `-i` short flags can be combined to perform a recursive, case-insensitive search of one or more directories:

```
$ grep -ri the .
./nobody.txt:Then there's a pair of us!
./nobody.txt:Don't tell! they'd advertise—you know!
./nobody.txt:To tell one's name—the livelong June—
./bustle.txt:The bustle in a house
./bustle.txt:The morning after death
./bustle.txt:The sweeping up the heart,
./fox.txt:The quick brown fox jumps over the lazy dog.
```

Without any positional arguments for inputs, `grep` will read `STDIN`:

```
$ cat * | grep -i the
The bustle in a house
The morning after death
The sweeping up the heart,
The quick brown fox jumps over the lazy dog.
Then there's a pair of us!
Don't tell! they'd advertise—you know!
To tell one's name—the livelong June—
```

This is as far as the challenge program is expected to go.

Getting Started

The name of the challenge program should be `grepr` for a Rust version of `grep`. Start with **`cargo new grepr`**, then copy the *09_grepr/tests* directory into your new project. In addition to using `clap` to parse the command-line arguments, my solution will use `regex` for regular expressions and `walkdir` to find the input files. Here is how I start the *Cargo.toml*:

```
[dependencies]
clap = "2.33"
regex = "1"
walkdir = "2"
sys-info = "0.9"

[dev-dependencies]
assert_cmd = "1"
predicates = "1"
rand = "0.8"
```

You can run **`cargo test`** to perform an initial build and run the tests, all of which should fail.

Defining the Arguments

I will start with the following for *src/main.rs*:

```
fn main() {
    if let Err(e) = grepr::get_args().and_then(grepr::run) {
        eprintln!("{}", e);
        std::process::exit(1);
    }
}
```

Following is how I started my *src/lib.rs*. Note that all the Boolean options default to `false`:

```
use clap::{App, Arg};
use regex::{Regex, RegexBuilder};
use std::error::Error;

type MyResult<T> = Result<T, Box<dyn Error>>;
```

```
#[derive(Debug)]
pub struct Config {
    pattern: Regex, ❶
    files: Vec<String>, ❷
    recursive: bool, ❸
    count: bool, ❹
    invert_match: bool, ❺
}
```

❶ The `pattern` is a compiled regular expression.

❷ The `files` is a vector of strings.

❸ The `recursive` option is a Boolean to recursively search directories.

❹ The `count` option is a Boolean to display a count of the matches.

❺ The `invert_match` is a Boolean to find lines that do not match the pattern.

Here is how I started my `get_args` and `run` functions. You should fill in the missing parts:

```
pub fn get_args() -> MyResult<Config> {
    let matches = App::new("grepr")
        .version("0.1.0")
        .author("Ken Youens-Clark <kyclark@gmail.com>")
        .about("Rust grep")
        // What goes here?
        .get_matches();

    Ok(Config {
        pattern: ...
        files: ...
        recursive: ...
        count: ...
        invert_match: ...
    })
}

pub fn run(config: Config) -> MyResult<()> {
    println!("{:#?}", config);
    Ok(())
}
```

Your program should be able to produce the following usage:

```
grepr 0.1.0
Ken Youens-Clark <kyclark@gmail.com>
Rust grep

USAGE:
    grepr [FLAGS] <PATTERN> <FILE>...

FLAGS:
    -c, --count           Count occurrences
    -h, --help            Prints help information
    -i, --insensitive     Case-insensitive
    -v, --invert-match    Invert match
    -r, --recursive       Recursive search
    -V, --version         Prints version information

ARGS:
    <PATTERN>    Search pattern ❶
    <FILE>...    Input file(s) [default: -] ❷
```

❶ The search pattern is a required argument.

❷ The input files are optional and default to "-" for `STDIN`.

Your program should be able to print a `Config` like the following when provided a pattern and no input files:

```
$ cargo run -- dog
Config {
    pattern: dog,
    files: [
        "-",
    ],
    recursive: false,
    count: false,
    invert_match: false,
}
```

It should be able to handle one or more input files and handle the flags:

```
$ cargo run -- dog -ricv tests/inputs/*.txt
Config {
```

```
    pattern: dog,
    files: [
        "tests/inputs/bustle.txt",
        "tests/inputs/empty.txt",
        "tests/inputs/fox.txt",
        "tests/inputs/nobody.txt",
    ],
    recursive: true,
    count: true,
    invert_match: true,
}
```

It should reject an invalid regular expression. For instance, `*` signifies *zero or more* of the preceding pattern. By itself, this is incomplete:

```
$ cargo run -- \*
Invalid pattern "*"
```

I assume you figured that out. Following is how I declared my arguments:

```
pub fn get_args() -> MyResult<Config> {
    let matches = App::new("grepr")
        .version("0.1.0")
        .author("Ken Youens-Clark <kyclark@gmail.com>")
        .about("Rust grep")
        .arg(
            Arg::with_name("pattern") ❶
                .value_name("PATTERN")
                .help("Search pattern")
                .required(true),
        )
        .arg(
            Arg::with_name("files") ❷
                .value_name("FILE")
                .help("Input file(s)")
                .required(true)
                .default_value("-")
                .min_values(1),
        )
        .arg(
            Arg::with_name("insensitive") ❸
                .value_name("INSENSITIVE")
                .help("Case-insensitive")
                .short("i")
                .long("insensitive")
```

```
            .takes_value(false),
    )
    .arg(
        Arg::with_name("recursive") ❹
            .value_name("RECURSIVE")
            .help("Recursive search")
            .short("r")
            .long("recursive")
            .takes_value(false),
    )
    .arg(
        Arg::with_name("count") ❺
            .value_name("COUNT")
            .help("Count occurrences")
            .short("c")
            .long("count")
            .takes_value(false),
    )
    .arg(
        Arg::with_name("invert") ❻
            .value_name("INVERT")
            .help("Invert match")
            .short("v")
            .long("invert-match")
            .takes_value(false),
    )
    .get_matches();
```

❶ The first positional argument is for the `pattern`.

❷ The rest of the positional arguments are for the inputs. The default is "-".

❸ The `insensitive` flag will handle case-insensitive options.

❹ The `recursive` flag will handle searching for files in directories.

❺ The `count` flag will cause the program to print counts.

❻ The `invert` flag will search for lines not matching the pattern.

NOTE

Here the order in which you declare the positional parameters is important as the first one defined will be for the first positional argument. You can still define the options before or after the positional parameters.

Next, I used the arguments to create a regular expression that will incorporate the `insensitive` option:

```
    let pattern = matches.value_of("pattern").unwrap(); ❶
    let pattern = RegexBuilder::new(pattern) ❷
        .case_insensitive(matches.is_present("insensitive")) ❸
        .build() ❹
        .map_err(|_| format!("Invalid pattern \"{}\"", pattern))?;
❺

    Ok(Config { ❻
        pattern,
        files: matches.values_of_lossy("files").unwrap(),
        recursive: matches.is_present("recursive"),
        count: matches.is_present("count"),
        invert_match: matches.is_present("invert"),
    })
}
```

❶ The `pattern` is required, so it should be safe to unwrap the value.

❷ The `RegexBuilder::new` method will create a new regular expression.

❸ The `Regex::case_insensitive` method will cause the regex to disregard case in comparisons when the `insensitive` is flag is present.

❹ The `Regex::build` method will compile the regex.

❺ If `build` returns an error, use `Result::map_err` to create an error message that the given pattern is invalid.

❻ Return the `Config`.

`RegexBuilder::build` will reject any pattern that is not a valid regular expression, and this raises an interesting point. There are many syntaxes for writing regular expressions. If you look closely at the manual page for `grep`, you'll notice these options:

```
-E, --extended-regexp
        Interpret pattern as an extended regular expression (i.e.
```

```
force
        grep to behave as egrep).

-e pattern, --regexp=pattern
        Specify a pattern used during the search of the input: an
input
        line is selected if it matches any of the specified
patterns.
        This option is most useful when multiple -e options are
used to
        specify multiple patterns, or when a pattern begins with a
dash
        (`-').
```

The converse of these options is:

```
-G, --basic-regexp
        Interpret pattern as a basic regular expression (i.e. force
grep
        to behave as traditional grep).
```

Regular expressions have been around since the 1950s when they were invented by the American mathematician Stephen Cole Kleene[2]. Since that time, the syntax has been modified and expanded by various groups, perhaps most notably by the Perl community which created Perl Compatible Regular Expressions (PCRE). By default, `grep` will only parse basic regexes, but the preceding flags can allow it to use other varieties. For instance, I can use the pattern `ee` to search for any lines containing two adjacent *e*s:

```
$ grep 'ee' tests/inputs/*
tests/inputs/bustle.txt:The sweeping up the heart,
```

If I wanted to find any character that is repeated twice, the pattern is `(.)\1` where the dot (`.`) represents any character and the capturing parentheses allow me to use the backreference `\1` to refer to the first capture group. This is an example of an extended expression, and so requires the `-E` flag:

```
$ grep -E '(.)\1' tests/inputs/*
tests/inputs/bustle.txt:The sweeping up the heart,
tests/inputs/bustle.txt:And putting love away
```

```
tests/inputs/bustle.txt:We sha[ll] not want to use again
tests/inputs/nobody.txt:Are you—Nobody—t[oo]?
tests/inputs/nobody.txt:Don't te[ll]! they'd advertise—you know!
tests/inputs/nobody.txt:To te[ll] one's name—the livelong June—
```

The Rust `regex` crate documentation notes that its "syntax is similar to Perl-style regular expressions, but lacks a few features like look around and backreferences." (*Look-around* assertions allow the expression to assert that a pattern must be followed or preceded by another pattern, and *backreferences* allow the pattern to refer to previously captured values.) This means that the challenge program will work more like the `egrep` in handling extended regular expressions by default. Sadly, this also means that the program will not be able to handle the preceding pattern because it requires backreferences. It will still be a wicked cool program to write, so let's get at it.

Finding the Files to Search

To use the compiled regex, I next need to find all the files to search. Recall that the user might provide directory names with the `--recursive` option to search for all the files contained in each directory; otherwise, directory names should result in a warning printed to `STDERR`. I decided to write a function called `find_files` that will accept a vector of strings which may be file or directory names along with a Boolean for whether or not to recurse into directories. It returns a vector of `MyResult` values that will either hold a string which is the name of a valid file or an error message:

```
fn find_files(files: &[String], recursive: bool) ->
Vec<MyResult<String>> {
    unimplemented!();
}
```

To test this, I can add a `tests` module to *src/lib.rs*. Note that this will use the `rand` module which should be listed in the `[dev-dependencies]` section of your *Cargo.toml* as noted earlier in the chapter:

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

```
    use super::find_files;
    use rand::{distributions::Alphanumeric, Rng};

    #[test]
    fn test_find_files() {
        // Verify that the function finds a file known to exist
        let files =
            find_files(&["./tests/inputs/fox.txt".to_string()],
false);
        assert_eq!(files.len(), 1);
        assert_eq!(files[0].as_ref().unwrap(),
"./tests/inputs/fox.txt");

        // The function should reject a directory without the
recursive option
        let files = find_files(&["./tests/inputs".to_string()],
false);
        assert_eq!(files.len(), 1);
        if let Err(e) = &files[0] {
            assert_eq!(
                e.to_string(),
                "./tests/inputs is a directory".to_string()
            );
        }

        // Verify the function recurses to find four files in the
directory
        let res = find_files(&["./tests/inputs".to_string()],
true);
        let mut files: Vec<String> = res
            .iter()
            .map(|r| r.as_ref().unwrap().replace("\\", "/"))
            .collect();
        files.sort();
        assert_eq!(files.len(), 4);
        assert_eq!(
            files,
            vec![
                "./tests/inputs/bustle.txt",
                "./tests/inputs/empty.txt",
                "./tests/inputs/fox.txt",
                "./tests/inputs/nobody.txt",
            ]
        );

        // Generate a random string to represent a nonexistent file
        let bad: String = rand::thread_rng()
            .sample_iter(&Alphanumeric)
```

```
            .take(7)
            .map(char::from)
            .collect();

        // Verify that the function returns the bad file as an
error
        let files = find_files(&[bad], false);
        assert_eq!(files.len(), 1);
        assert!(files[0].is_err());
    }
}
```

You should be able to run **`cargo test test_find_files`** to verify that your function finds existing files, recurses properly, and will report nonexistent files as errors. Here is how I can use it in my code:

```
pub fn run(config: Config) -> MyResult<()> {
    println!("pattern \"{}\"", config.pattern);

    for entry in find_files(&config.files, config.recursive) {
        match entry {
            Err(e) => eprintln!("{}", e),
            Ok(filename) => println!("file \"{}\"", filename),
        }
    }

    Ok(())
}
```

My solution uses `WalkDir`, which I introduced in Chapter 7 (`findr`). See if you can get your program to reproduce the following output. To start, the default input should be "-" to represent reading from `STDIN`:

```
$ cargo run -- fox
pattern "fox"
file "-"
```

NOTE

Printing a regular expression means calling the `Regex::as_str` method. `Regex::build` notes that this "will produce the pattern given to new verbatim. Notably, it will not incorporate any of the flags set on this builder."

Explicitly listing “-” as the input should produce the same output:

```
$ cargo run -- fox -
pattern "fox"
file "-"
```

The program should handle multiple input files:

```
$ cargo run -- fox tests/inputs/*
pattern "fox"
file "tests/inputs/bustle.txt"
file "tests/inputs/empty.txt"
file "tests/inputs/fox.txt"
file "tests/inputs/nobody.txt"
```

A directory name without a `--recursive` option should be rejected:

```
$ cargo run -- fox tests/inputs
pattern "fox"
tests/inputs is a directory
```

With the `--recursive` flag, it should find the directory’s files:

```
$ cargo run -- -r fox tests/inputs
pattern "fox"
file "tests/inputs/empty.txt"
file "tests/inputs/nobody.txt"
file "tests/inputs/bustle.txt"
file "tests/inputs/fox.txt"
```

Nonexistent arguments should be printed to `STDERR` in the course of handling each entry:

```
$ cargo run -- -r fox blargh tests/inputs/fox.txt
pattern "fox"
blargh: No such file or directory (os error 2)
file "tests/inputs/fox.txt"
```

Finding the Matching Lines of Input

Once you are properly handling the inputs, it's time to open the files and search for matching lines. I suggest you again use the `open` function from earlier chapters that will open and read either an existing file or `STDIN` for the filename "-". You will need to add `use std::fs::File` and `use std::io::{self, BufRead, BufReader}` for this:

```
fn open(filename: &str) -> MyResult<Box<dyn BufRead>> {
    match filename {
        "-" => Ok(Box::new(BufReader::new(io::stdin()))),
        _ => Ok(Box::new(BufReader::new(File::open(filename)?))),
    }
}
```

When reading the lines, be sure to preserve the line endings as one of the input files contains Windows-style CRLF endings. My solution uses a function called `find_lines` which you can start with the following:

```
fn find_lines<T: BufRead>(
    mut file: T, ❶
    pattern: &Regex, ❷
    invert_match: bool, ❸
) -> MyResult<Vec<String>> {
    unimplemented!();
}
```

❶ The `file` must implement the `std::io::BufRead` trait.

❷ The `pattern` is a reference to a compiled regular expression.

❸ The `invert_match` is a Boolean for whether to reverse the match operation.

> **NOTE**
>
> In Chapter 5, I used `impl BufRead` to indicate a value that must implement the `BufRead`. In the preceding code, I'm using `<T: BufRead>` to write the trait bound for the type `T`.

To test this function, I expanded my `tests` module by adding the following `test_find_lines` function which again uses `std::io::Cursor` to create a fake filehandle that implements `BufRead` for testing:

```
#[cfg(test)]
mod test {
    use super::{find_files, find_lines};
    use rand::{distributions::Alphanumeric, Rng};
    use regex::{Regex, RegexBuilder};
    use std::io::Cursor;

    #[test]
    fn test_find_lines() {
        let text = b"Lorem\nIpsum\r\nDOLOR";

        // The pattern _or_ should match the one line, "Lorem"
        let re1 = Regex::new("or").unwrap();
        let matches = find_lines(Cursor::new(&text), &re1, false);
        assert!(matches.is_ok());
        assert_eq!(matches.unwrap().len(), 1);

        // When inverted, the function should match the other two
lines
        let matches = find_lines(Cursor::new(&text), &re1, true);
        assert!(matches.is_ok());
        assert_eq!(matches.unwrap().len(), 2);

        // This regex will be case-insensitive
        let re2 = RegexBuilder::new("or") ❸
            .case_insensitive(true)
            .build()
            .unwrap();

        // The two lines "Lorem" and "DOLOR" should match
        let matches = find_lines(Cursor::new(&text), &re2, false);
❹
        assert!(matches.is_ok());
        assert_eq!(matches.unwrap().len(), 2);

        // When inverted, the one remaining line should match
        let matches = find_lines(Cursor::new(&text), &re2, true);
        assert!(matches.is_ok());
        assert_eq!(matches.unwrap().len(), 1);
    }

    #[test]
```

```
    fn test_find_files() {} // Same as before
}
```

Try writing this function and then running **`cargo test test_find_lines`** until it passes. Next, I suggest you incorporate these ideas into your `run`:

```
pub fn run(config: Config) -> MyResult<()> {
    let entries = find_files(&config.files, config.recursive); ❶
    for entry in entries {
        match entry {
            Err(e) => eprintln!("{}", e), ❷
            Ok(filename) => match open(&filename) { ❸
                Err(e) => eprintln!("{}: {}", filename, e), ❹
                Ok(_file) => println!("Opened {}", filename), ❺
            },
        }
    }

    Ok(())
}
```

❶ Look for the input files.

❷ Handle the errors from finding input files.

❸ Try to open a valid filename.

❹ Handle errors opening a file.

❺ Here you have an open filehandle.

Start as simply as possible, perhaps by using an empty regular expression that should match all the lines from the input:

```
$ cargo run -- "" tests/inputs/fox.txt
The quick brown fox jumps over the lazy dog.
```

Be sure you are reading `STDIN` by default:

```
$ cargo run -- "" < tests/inputs/fox.txt
The quick brown fox jumps over the lazy dog.
```

Run with several input files and a case-sensitive pattern:

```
$ cargo run -- The tests/inputs/*
tests/inputs/bustle.txt:The bustle in a house
tests/inputs/bustle.txt:The morning after death
tests/inputs/bustle.txt:The sweeping up the heart,
tests/inputs/fox.txt:The quick brown fox jumps over the lazy dog.
tests/inputs/nobody.txt:Then there's a pair of us!
```

Then try to print the number of matches instead of the lines:

```
$ cargo run -- --count The tests/inputs/*
tests/inputs/bustle.txt:3
tests/inputs/empty.txt:0
tests/inputs/fox.txt:1
tests/inputs/nobody.txt:1
```

Incorporate the `--insensitive` option:

```
$ cargo run -- --count --insensitive The tests/inputs/*
tests/inputs/bustle.txt:3
tests/inputs/empty.txt:0
tests/inputs/fox.txt:1
tests/inputs/nobody.txt:3
```

Next, try to invert the matching:

```
$ cargo run -- --count --invert-match The tests/inputs/*
tests/inputs/bustle.txt:6
tests/inputs/empty.txt:0
tests/inputs/fox.txt:0
tests/inputs/nobody.txt:8
```

Be sure your `--recursive` option works:

```
$ cargo run -- -icr the tests/inputs
tests/inputs/empty.txt:0
tests/inputs/nobody.txt:3
tests/inputs/bustle.txt:3
tests/inputs/fox.txt:1
```

Handle errors like nonexistent files while processing the files in order:

```
$ cargo run -- fox blargh tests/inputs/fox.txt
blargh: No such file or directory (os error 2)
tests/inputs/fox.txt:The quick brown fox jumps over the lazy dog.
```

Another potential problem you should gracefully handle is a failure to open a file perhaps due to insufficient permissions:

```
$ touch hammer && chmod 000 hammer
$ cargo run -- fox hammer tests/inputs/fox.txt
hammer: Permission denied (os error 13)
tests/inputs/fox.txt:The quick brown fox jumps over the lazy dog.
```

These challenges are getting harder, so it's OK to feel a bit overwhelmed by the requirements. Try to tackle each task in order, and keep running **`cargo test`** to see how many you're able to pass. When you get stuck, run `grep` with the arguments and closely examine the output. Then run your program with the same arguments and try to find the differences.

Solution

To start, I'll share my `find_files` function:

```
fn find_files(files: &[String], recursive: bool) ->
Vec<MyResult<String>> {
    let mut results = vec![]; ❶

    for path in files { ❷
        match path.as_str() {
            "-" => results.push(Ok(path.to_string())), ❸
            _ => match fs::metadata(&path) { ❹
                Ok(metadata) => {
                    if metadata.is_dir() { ❺
                        if recursive { ❻
                            for entry in WalkDir::new(path) ❼
                                .into_iter()
                                .filter_map(|e| e.ok())
                                .filter(|e|
e.file_type().is_file())
```

```
                            {
                                results.push(Ok(entry
                                    .path()
                                    .display()
                                    .to_string()));
                            }
                        } else {
                            results.push(Err(From::from(format!( ❽
                                "{} is a directory",
                                path
                            ))));
                        }
                    } else if metadata.is_file() { ❾
                        results.push(Ok(path.to_string()));
                    }
                }
                Err(e) => { ❿
                    results.push(Err(From::from(format!("{}: {}",
path, e))))
                }
            },
        }
    }

    results
}
```

❶ Initialize an empty vector to hold the `results`.

❷ Iterate over each of the given filenames.

❸ First, accept the filename “-” for `STDIN`.

❹ Try to get the file’s metadata.

❺ Check if the entry is a directory.

❻ Check if the user wants to recursively search directories.

❼ Add all the files in the given directory to the `results`.

❽ Note an error that the given entry is a directory.

❾ If the entry is a file, add it to the `results`.

❿ This arm will be triggered by nonexistent files.

Next, I will share my `find_lines` function. This borrows heavily from

previous functions that read files line-by-line, so I won't comment on code I've used before:

```
fn find_lines<T: BufRead>(
    mut file: T,
    pattern: &Regex,
    invert_match: bool,
) -> MyResult<Vec<String>> {
    let mut matches = vec![]; ❶
    let mut line = String::new();

    loop {
        let bytes = file.read_line(&mut line)?;
        if bytes == 0 {
            break;
        }
        if (pattern.is_match(&line) && !invert_match) ❷
            || (!pattern.is_match(&line) && invert_match) ❸
        {
            matches.push(line.clone()); ❹
        }
        line.clear();
    }

    Ok(matches) ❸
}
```

❶ Initialize a mutable vector to hold the matching lines.

❷ Verify that the lines matches and I'm not supposed to invert the match.

❸ Alternately if the line does not match and I am supposed to invert the match.

❹ I must `clone` the string to add it to the `matches`.

> **NOTE**
>
> In the preceding function, the `&&` is a short-circuiting logical *AND* that will only evaluate to `true` if *both* operands are `true`. The `||` is the short-circuiting logical *OR*, and will evaluate to `true` if *either* of the operands is `true`.

At the beginning of my `run` function, I decided to create a closure to handle the printing of the output with or without the filenames given the number of input files:

```
pub fn run(config: Config) -> MyResult<()> {
    let entries = find_files(&config.files, config.recursive); ❶
    let num_files = &entries.len(); ❷
    let print = |fname: &str, val: &str| { ❸
        if num_files > &1 {
            print!("{}:{}", fname, val);
        } else {
            print!("{}", val);
        }
    };
```

❶ Find all the inputs.

❷ Find the number of inputs.

❸ Create a `print` closure that uses the number of inputs to decide whether to print the filename in the output.

Continuing from there, my code attempts to find the matching lines from the entries:

```
    for entry in entries {
        match entry {
            Err(e) => eprintln!("{}", e), ❶
            Ok(filename) => match open(&filename) { ❷
                Err(e) => eprintln!("{}: {}", filename, e), ❸
                Ok(file) => {
                    match find_lines( ❹
                        file,
                        &config.pattern,
                        config.invert_match,
                    ) {
                        Err(e) => eprintln!("{}", e), ❺
                        Ok(matches) => {
                            if config.count { ❻
                                print(
                                    &filename,
                                    &format!("{}\n",
&matches.len()),
```

```
                                );
                            } else {
                                for line in &matches {
                                    print(&filename, line);
                                }
                            }
                        }
                    }
                },
            }
        }
        Ok(())
    }
```

1. Print errors like nonexistent files to `STDERR`.
2. Attempt to open an existing file that could fail because of permissions.
3. Print an error to `STDERR`.
4. Attempt to find the matching lines of text.
5. Print errors to `STDERR`.
6. Decide whether to print the number of matches or the matches themselves.

Going Further

`ripgrep` is a very complete Rust implementation of `grep` and is well worthy of your study. You can install the program using the instructions provided and then execute `rg`. As shown in Figure 9-1, the matching text is highlighted in the ouput. Try to add that feature to your program using `Regex::find` to find the start and stop positions of the matching pattern and something like `colorize` to highlight the match.

```
$ rg The tests/inputs/*
tests/inputs/fox.txt
1:The quick brown fox jumps over the lazy dog.

tests/inputs/nobody.txt
3:Then there's a pair of us?

tests/inputs/bustle.txt
1:The bustle in a house
2:The morning after death
6:The sweeping up the heart,
```

Figure 9-1. The `rg` tool will highlight the matching text

Summary

Mostly this chapter challenged you to extend skills you've learned from previous chapters. For instance, in Chapter 7 (`findr`), you learned how to recursively find files in directories, and several previous chapters have used regular expressions. In this chapter, you combined those skills to find content in files matching (or not) a given regex.

In addition, you learned the following:

- How to use `RegexBuilder` to create more complicated regular expressions using, for instance, the case-insensitive option to match strings regardless of case.
- There are multiple syntaxes for writing regular expressions that different tools recognize such as PCRE. Rust's `regex` engine does not implement some features of PCRE such as look-around asssertions or backreferences.
- You can indicate the trait bound `BufRead` in function signatures either using `impl BufRead` or by using `<T: BufRead>`.

- Rust's logical *AND* operator (&&) evaluates to `true` if *both* the operands are `true`. The *OR* (`||`) operator evaluates to `true` is *either* is `true`.

---

1 The name `grep` comes from the `sed` command `g/re/p` which means *global regular expression print*.

2 If you would like to learn more about regexes, I recommend *Mastering Regular Expressions* by Jeffrey Friedl (O'Reilly, 2006).

Chapter 10. Boston Commons

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 10th Chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the author at kyclark@gmail.com.

Can you fault such common sense?

—They Might Be Giants

The `comm` (*common*) utility will read two files and report the lines of text that are common to both and the lines that are unique to each. These are set operations where the common lines are the *intersection* of the two files and the unique lines are the *difference*. If you are familiar with databases, you might also consider these as types of *join* operations. In this chapter, you will learn:

- How to manually iterate the lines of a filehandle
- How to `match` on combinations of possibilities using a tuple
- How to compare two strings
- How to use `std::cmp::Ordering`
- How to create an `enum` variant that contains a value

How comm Works

I'll start with part of the manual page for the BSD `comm`:

```
COMM(1)                   BSD General Commands Manual
COMM(1)

NAME
     comm -- select or reject lines common to two files

SYNOPSIS
     comm [-123i] file1 file2

DESCRIPTION
     The comm utility reads file1 and file2, which should be sorted
lexically,
     and produces three text columns as output: lines only in
file1; lines
     only in file2; and lines in both files.

     The filename ''-'' means the standard input.

     The following options are available:

     -1      Suppress printing of column 1.

     -2      Suppress printing of column 2.

     -3      Suppress printing of column 3.

     -i      Case insensitive comparison of lines.

     Each column will have a number of tab characters prepended to
it equal to
     the number of lower numbered columns that are being printed.
For exam-
     ple, if column number two is being suppressed, lines printed
in column
     number one will not have any tabs preceding them, and lines
printed in
     column number three will have one.

     The comm utility assumes that the files are lexically sorted;
all charac-
     ters participate in line comparisons.
```

It might help to see some examples:

```
EXAMPLES
     Assuming a file named example.txt with the following contents:

           a
           b
           c
           d

     Show lines only in example.txt, lines only in stdin and common
lines:

           $ echo -e "B\nc" | comm example.txt -
                   B
           a
           b
                          c
           d

     Show only common lines doing case insensitive comparisons:

           $ echo -e "B\nc" | comm -1 -2 -i example.txt -
           b
           c
```

The GNU version has some additional options:

```
$ comm --help
Usage: comm [OPTION]... FILE1 FILE2
Compare sorted files FILE1 and FILE2 line by line.

When FILE1 or FILE2 (not both) is -, read standard input.

With no options, produce three-column output.  Column one contains
lines unique to FILE1, column two contains lines unique to FILE2,
and column three contains lines common to both files.

  -1              suppress column 1 (lines unique to FILE1)
  -2              suppress column 2 (lines unique to FILE2)
  -3              suppress column 3 (lines that appear in both
files)

  --check-order     check that the input is correctly sorted, even
                      if all input lines are pairable
  --nocheck-order   do not check that the input is correctly sorted
```

```
  --output-delimiter=STR  separate columns with STR
  --total           output a summary
  -z, --zero-terminated    line delimiter is NUL, not newline
      --help     display this help and exit
      --version  output version information and exit

Note, comparisons honor the rules specified by 'LC_COLLATE'.

Examples:
  comm -12 file1 file2  Print only lines present in both file1 and
file2.
  comm -3 file1 file2  Print lines in file1 not in file2, and vice
versa.
```

At this point, you may be wondering exactly why you'd use this. Consider that you have a list of cities where your favorite band played on their last tour:

```
$ cd 10_commr/tests/inputs/
$ cat cities1.txt
Jackson
Denton
Cincinnati
Boston
Santa Fe
Tucson
```

Another file lists the cities on their current tour:

```
$ cat cities2.txt
San Francisco
Denver
Ypsilanti
Denton
Cincinnati
Boston
```

You can use `comm` to find which cities occur in both sets by suppressing columns 1 (the lines unique to the first file) and 2 (the lines unique to the second file) and only show column 3 (the lines common to both files). This is like an *inner join* in SQL where only data that occurs in both inputs is shown. Note that both files need to be sorted first:

```
$ comm -12 <(sort cities1.txt) <(sort cities2.txt)
Boston
Cincinnati
Denton
```

If you wanted the cities they only played on the first tour, you could suppress columns 2 and 3:

```
$ comm -23 <(sort cities1.txt) <(sort cities2.txt)
Jackson
Santa Fe
Tucson
```

Finally, if you wanted the cities they only played on the second tour, you could suppress columns 1 and 3:

```
$ comm -13 <(sort cities1.txt) <(sort cities2.txt)
Denver
San Francisco
Ypsilanti
```

The first or second file can be `STDIN` as denoted by the filename "-":

```
$ sort cities2.txt | comm -12 <(sort cities1.txt) -
Boston
Cincinnati
Denton
```

As with the GNU `comm`, only one of the inputs may be "-" with the challenge program. Note that the `comm` can perform case-insensitive comparisons when the `-i` flag is present. For instance, I can put the first tour cities in lowercase:

```
$ cat cities1_lower.txt
jackson
denton
cincinnati
boston
santa fe
tucson
```

and the second tour cities in uppercase:

```
$ cat cities2_upper.txt
SAN FRANCISCO
DENVER
YPSILANTI
DENTON
CINCINNATI
BOSTON
```

Then I can use the `-i` flag to find the cities in common:

```
$ comm -i -12 <(sort cities1_lower.txt) <(sort cities2_upper.txt)
boston
cincinnati
denton
```

> **NOTE**
>
> I know the tour cities example is a trivial one, so I'll give you another example drawn from my experience in bioinformatics, which is the intersection of computer science and biology. Given a file of protein sequences, I can run an analysis that will group similar sequences into clusters. I can then use `comm` to compare the clustered proteins to the original list and find the proteins that failed to cluster. There may be something unique to these unclustered proteins that bears further analysis.

This is as much as the challenge program is expected to implement. One change from the original tool is that I wanted to add an optional output column delimiter that defaults to a tab character, which is the normal output from `comm`.

Getting Started

The program in this chapter will be called `commr` (pronounced *comm-ar*) for a Rust version of `comm`. I suggest you use **`cargo new commr`** to start, then add the following dependencies to your *Cargo.toml* file:

```
[dependencies]
clap = "2.33"
```

```
[dev-dependencies]
assert_cmd = "1"
predicates = "1"
rand = "0.8"
```

Copy my *10_commr/tests* directory into your project, and then run **`cargo test`** to run the tests which should all fail.

Defining the Arguments

No surprises here, but I suggest the following for your *src/main.rs*:

```
fn main() {
    if let Err(e) = commr::get_args().and_then(commr::run) {
        eprintln!("{}", e);
        std::process::exit(1);
    }
}
```

You can start *src/lib.rs* with the following definition for `Config`:

```
use clap::{App, Arg};
use std::error::Error;

type MyResult<T> = Result<T, Box<dyn Error>>;

#[derive(Debug)]
pub struct Config {
    file1: String, ❶
    file2: String, ❷
    suppress_col1: bool, ❸
    suppress_col2: bool, ❹
    suppress_col3: bool, ❺
    insensitive: bool, ❻
    delimiter: String, ❼
}
```

❶ The first input filename is a `String`.

❷ The second input filename is a `String`.

❸ A Boolean for whether to suppress the first column of output.

❹ A Boolean for whether to suppress the second column of output.

❺ A Boolean for whether to suppress the third column of output.

❻ A Boolean for whether to perform case-insensitive comparisons.

❼ The output column delimiter, which will default to a tab.

You can start your `get_args` like so:

```
pub fn get_args() -> MyResult<Config> {
    let matches = App::new("commr")
        .version("0.1.0")
        .author("Ken Youens-Clark <kyclark@gmail.com>")
        .about("Rust comm")
        // What goes here?
        .get_matches();

        Ok(Config {
        file1: ...
        file2: ...
        suppress_col1: ...
        suppress_col2: ...
        suppress_col3: ...
        insensitive: ...
        delimiter: ...
    })
}
```

Start your `run` function by printing the `config`:

```
pub fn run(config: Config) -> MyResult<()> {
    println!("{:#?}", config);
    Ok(())
}
```

Your program should be able to produce the following usage:

```
cargo run -- -h
commr 0.1.0
Ken Youens-Clark <kyclark@gmail.com>
Rust comm

USAGE:
```

```
    commr [FLAGS] [OPTIONS] <FILE1> <FILE2>

FLAGS:
    -h, --help       Prints help information
    -i               Case insensitive comparison of lines
    -1               Suppress printing of column 1
    -2               Suppress printing of column 2
    -3               Suppress printing of column 3
    -V, --version    Prints version information

OPTIONS:
    -d, --output-delimiter <DELIM>    Output delimiter

ARGS:
    <FILE1>    Input file 1
    <FILE2>    Input file 2
```

If you run your program with no arguments, it should fail with a message that the two file arguments are required:

```
$ cargo run
error: The following required arguments were not provided:
    <FILE1>
    <FILE2>

USAGE:
    commr [FLAGS] [OPTIONS] <FILE1> <FILE2>

For more information try --help
```

If you supply two positional arguments, you should get the following output:

```
$ cargo run -- tests/inputs/file1.txt tests/inputs/file2.txt
Config {
    file1: "tests/inputs/file1.txt", ❶
    file2: "tests/inputs/file2.txt",
    suppress_col1: false, ❷
    suppress_col2: false,
    suppress_col3: false,
    insensitive: false,
    delimiter: "\t",
}
```

❶ The two positional arguments are parsed into `file1` and `file2`.

❷ All the rest of the values use defaults which are `false` for the Booleans and the tab character for `delimiter`.

Verify that you can set all the other arguments as well:

```
$ cargo run -- tests/inputs/file1.txt tests/inputs/file2.txt -123 -
d , -i
Config {
    file1: "tests/inputs/file1.txt",
    file2: "tests/inputs/file2.txt",
    suppress_col1: true, ❶
    suppress_col2: true,
    suppress_col3: true,
    insensitive: true, ❷
    delimiter: ",", ❸
}
```

❶ The `-123` sets each of the *suppress* values to `true`.

❷ The `-i` sets `insensitive` to `true`.

❸ The `-d` option sets the `delimiter` to a comma (`,`).

Stop reading and make your program match the preceding output. Come back when you're done.

Following is how I defined the arguments in my `get_args`. I don't have much to comment on here since it's so similar to previous programs:

```
pub fn get_args() -> MyResult<Config> {
    let matches = App::new("commr")
        .version("0.1.0")
        .author("Ken Youens-Clark <kyclark@gmail.com>")
        .about("Rust comm")
        .arg(
            Arg::with_name("file1")
                .value_name("FILE1")
                .help("Input file 1")
                .takes_value(true)
                .required(true),
        )
        .arg(
            Arg::with_name("file2")
```

```
                .value_name("FILE2")
                .help("Input file 2")
                .takes_value(true)
                .required(true),
        )
        .arg(
            Arg::with_name("suppress_col1")
                .short("1")
                .value_name("COL1")
                .takes_value(false)
                .help("Suppress printing of column 1"),
        )
        .arg(
            Arg::with_name("suppress_col2")
                .short("2")
                .value_name("COL2")
                .takes_value(false)
                .help("Suppress printing of column 2"),
        )
        .arg(
            Arg::with_name("suppress_col3")
                .short("3")
                .value_name("COL3")
                .takes_value(false)
                .help("Suppress printing of column 3"),
        )
        .arg(
            Arg::with_name("insensitive")
                .short("i")
                .value_name("INSENSITIVE")
                .takes_value(false)
                .help("Case insensitive comparison of lines"),
        )
        .arg(
            Arg::with_name("delimiter")
                .short("d")
                .long("output-delimiter")
                .value_name("DELIM")
                .help("Output delimiter")
                .takes_value(true),
        )
        .get_matches();

    Ok(Config {
        file1: matches.value_of("file1").unwrap().to_string(),
        file2: matches.value_of("file2").unwrap().to_string(),
        suppress_col1: matches.is_present("suppress_col1"),
        suppress_col2: matches.is_present("suppress_col2"),
```

```
        suppress_col3: matches.is_present("suppress_col3"),
        insensitive: matches.is_present("insensitive"),
        delimiter:
matches.value_of("delimiter").unwrap_or("\t").to_string(), ❶
    })
}
```

❶ Use `Option::unwrap_or` to unwrap the given value or use a default (a tab), then convert the value to a `String`.

At this point, your program should pass **`cargo test dies_no_args`**.

Validating and Opening the Input Files

The first order of business will be checking and opening the input files. I suggest a modification of the `open` function used in several previous chapters:

```
fn open(filename: &str) -> MyResult<Box<dyn BufRead>> {
    match filename {
        "-" => Ok(Box::new(BufReader::new(io::stdin()))),
        _ => Ok(Box::new(BufReader::new(
            File::open(filename)
                .map_err(|e| format!("{}: {}", filename, e))?, ❶
        ))),
    }
}
```

❶ Incorporate the `filename` into the error message.

This will require you to expand your imports accordingly:

```
use clap::{App, Arg};
use std::{
    error::Error,
    fs::File,
    io::{self, BufRead, BufReader},
};
```

As noted earlier, only one of the inputs is allowed to be "-" for `STDIN`. You

can use the following code for your `run` that will check the filenames and then open the files:

```
pub fn run(config: Config) -> MyResult<()> {
    let filename1 = &config.file1;
    let filename2 = &config.file2;

    if filename1.as_str() == "-" && filename2.as_str() == "-" { ❶
        return Err(From::from("Both input files cannot be STDIN
(\"-\")"));
    }

    let _file1 = open(&filename1)?; ❷
    let _file2 = open(&filename2)?;
    println!("Opened {} and {}", filename1, filename2); ❸

    Ok(())
}
```

❶ Check that both of the filenames are not "-".

❷ Attempt to open the two input files.

❸ Print a message so you know what happened.

Your program should reject two `STDIN` arguments:

```
$ cargo run -- - -
Both input files cannot be STDIN ("-")
```

It should be able to print the following for two good input files:

```
$ cargo run -- tests/inputs/file1.txt tests/inputs/file2.txt
Opened tests/inputs/file1.txt and tests/inputs/file2.txt
```

It should reject a bad file for either argument:

```
$ cargo run -- tests/inputs/file1.txt blargh
blargh: No such file or directory (os error 2)
```

You should pass all the tests for **`cargo test dies`**:

```
running 4 tests
test dies_both_stdin ... ok
test dies_no_args ... ok
test dies_bad_file1 ... ok
test dies_bad_file2 ... ok
```

Processing the Files

Now your program has validated all the arguments and opened the input files, either of which may be `STDIN`. Next, you need to iterate over the lines from each file to compare them. You can use `BufRead::lines` for this as it is not necessary to preserve line endings. Start simple, perhaps using the *empty.txt* file (which is empty) and *file1.txt* which should print all the lines from *file1.txt*:

```
$ cd tests/inputs/
$ comm file1.txt empty.txt
a
b
c
d
```

Ensure that you get the same output (but now in column 2) if you reverse the argument order:

```
$ comm empty.txt file1.txt
        a
        b
        c
        d
```

Next, try with *file1.txt* and *file2.txt*. Note that the following is the output from BSD `comm` and is the expected behavior for the challenge program:

```
$ comm file1.txt file2.txt
        B
a
b
                c
d
```

GNU `comm` uses a different ordering for which lines to show first when they are not equal:

```
$ comm file1.txt file2.txt
a
b
        B
                c
d
```

This is one of those programs that was always rather mysterious to me until I wrote my own version. I suggest you start by trying to read a line from each file. The documentation for `BufRead::lines` notes that it will return a `None` when it reaches the end of the file. Starting with the empty file as one of the arguments will force you to deal with having an uneven number of lines where you will have to advance one of the filehandles while the other stays the same. Then when you use two nonempty files, you'll have to consider how to read the files until you have matching lines and moving them independently, otherwise.

I'll see you on the flip side after you've written your solution.

Solution

Adding to the last version I showed, I decided to create iterators for the lines of each of the files that would incorporate a closure to handle case-insensitive comparisons:

```
pub fn run(config: Config) -> MyResult<()> {
    let filename1 = &config.file1;
    let filename2 = &config.file2;

    if filename1.as_str() == "-" && filename2.as_str() == "-" {
        return Err(From::from("Both input files cannot be STDIN
(\"-\")"));
    }

    let case = |line: String| { ❶
        if config.insensitive {
```

```
            line.to_lowercase()
        } else {
            line
        }
    };

    let mut lines1 =
        open(filename1)?.lines().filter_map(Result::ok).map(case);
❷
    let mut lines2 =
        open(filename2)?.lines().filter_map(Result::ok).map(case);

    let line1 = lines1.next(); ❸
    let line2 = lines2.next();
    println!("line1 = {:?}", line1); ❹
    println!("line2 = {:?}", line2);

    Ok(())
}
```

❶ Create a closure to lowercase each line of text when `config.insensitive` is `true`.

❷ Open the files, create a `lines` iterator that removes errors, and then map the lines through the `case` closure.

❸ Use `Iterator::next` to get the first line from each filehandle.

❹ Print the first two values.

> **NOTE**
>
> In the preceding code, I used the function `Result::ok` rather than writing a closure `|line| line.ok()`. They both accomplish the same thing, but the first is shorter.

As I suggested, I'll start with one of the files being empty:

```
$ cd ../..
$ cargo run -- tests/inputs/file1.txt tests/inputs/empty.txt
line1 = Some("a")
line2 = None
```

That led me to think about how I can move through the lines of each iterator based on the four different combinations of `Some(line)` and `None` that I can get from two iterators. In the following code, I place the possibilities inside a tuple, which is a “finite heterogeneous sequence” surrounded by parentheses:

```
let mut line1 = lines1.next(); ❶
let mut line2 = lines2.next();

loop {
    match (&line1, &line2) { ❷
        (Some(_val1), Some(_val2)) => { ❸
            line1 = lines1.next();
            line2 = lines2.next();
        }
        (Some(_val1), None) => { ❹
            line1 = lines1.next();
        }
        (None, Some(_val2)) => { ❺
            line2 = lines2.next();
        }
        (None, None) => break, ❻
    };
}
```

❶ Make the line variables mutable.

❷ Compare all possible combinations of the two *line* variables for two variants.

❸ When both are `Some` value, it’s possible to ask for the `next` line of each.

❹ When there is only the first value, only ask for the `next` line from the first file.

❺ Do the same for the second file.

❻ When there are no lines left in either file, break from the loop.

Next, I must compare the two values when I have both. When I only have a value from the first or second file, I should print those values in the first or second column, respectively. When they are the same, I need to print the value in column 3. When the first value is less than the second, I should print

the first value in column 1; otherwise, I should print the second value in column 2. To understand this last point, consider the following two input files which I'll place side by side so you can imagine how the code will read the lines:

```
$ cat tests/inputs/file1.txt
a
b
c
d
```

```
$ cat tests/inputs/file2.txt
B
c
```

To help you see the output from BSD `comm`, I will pipe the output into Perl to replace the tab characters with the string `--->` to make it clearer which columns are being printed:

```
$ comm tests/inputs/file1.txt tests/inputs/file2.txt | perl -pe
"s/\t/--->/g"
--->B
a
b
--->--->c
d
```

Now imagine your code reads the first line from each input and has *a* from *file1.txt* and *B* from *file2.txt*. They are not equal, so the question is which to print. The goal is to mimic BSD `comm`, so I know that the *B* should come first and be printed in the second column. When I compare *a* and *B*, I find that *a* is greater than *B* when they are ordered by their *code point* or numerical value. To help you see this, I've included a program in *util/ascii* that will show you a range of the ASCII table starting at the first printable character. Note that *a* has a value of 97 while *B* is 66:

```
 33: !   52: 4   71: G   90: Z  109: m
 34: "   53: 5   72: H   91: [  110: n
 35: #   54: 6   73: I   92: \  111: o
 36: $   55: 7   74: J   93: ]  112: p
 37: %   56: 8   75: K   94: ^  113: q
 38: &   57: 9   76: L   95: _  114: r
 39: '   58: :   77: M   96: `  115: s
 40: (   59: ;   78: N   97: a  116: t
```

```
41: )   60: <   79: O   98: b  117: u
42: *   61: =   80: P   99: c  118: v
43: +   62: >   81: Q  100: d  119: w
44: ,   63: ?   82: R  101: e  120: x
45: -   64: @   83: S  102: f  121: y
46: .   65: A   84: T  103: g  122: z
47: /   66: B   85: U  104: h  123: {
48: 0   67: C   86: V  105: i  124: |
49: 1   68: D   87: W  106: j  125: }
50: 2   69: E   88: X  107: k  126: ~
51: 3   70: F   89: Y  108: l  127: DEL
```

To mimic BSD `comm`, I should print the *lower* value (*B*) first and draw another value from that file for the next iteration. The GNU version does the opposite. Note you should add `use std::cmp::Ordering::*` to your imports for this code:

```
let mut line1 = lines1.next();
let mut line2 = lines2.next();

loop {
    match (&line1, &line2) {
        (Some(val1), Some(val2)) => match val1.cmp(val2) { ❶
            Equal => { ❷
                println!("{}", val1);
                line1 = lines1.next();
                line2 = lines2.next();
            }
            Less => { ❸
                println!("{}", val1);
                line1 = lines1.next();
            }
            _ => { ❹
                println!("{}", val2);
                line2 = lines2.next();
            }
        },
        (Some(val1), None) => {
            println!("{}", val1); ❺
            line1 = lines1.next();
        }
        (None, Some(val2)) => {
            println!("{}", val2); ❻
            line2 = lines2.next();
        }
```

```
            (None, None) => break,
        };
    }
```

1. Use `std::cmp` to compare the first value to the second. This will return a variant from `std::cmp::Ordering`.
2. When the two values are equal, print the first and get values from each of the files.
3. When the value from the first file is less than the value from the second file, print the first and request the next value from the first file.
4. Otherwise, print the value from the second file and request the next value from the second file.
5. When there is only a value from the first file, print it and continue requesting values from the first file.
6. When there is only a value from the second file, print it and continue requesting values from the second file.

If I run this code using a nonempty and empty file, it works:

```
$ cargo run -- tests/inputs/file1.txt tests/inputs/empty.txt
a
b
c
d
```

If I use *file1.txt* and *file2.txt*, it's not far from the expected output:

```
$ cargo run -- tests/inputs/file1.txt tests/inputs/file2.txt
B
a
b
c
d
```

I decided to create an enum called `Column` to represent in which column I should print a value. Each variant holds a `String`, and you can place the

following at the top of *src/lib.rs* near your `Config` declaration. Be sure to add `use crate::Column::*` to your import so you can reference `Col1` instead of `Column::Col1`:

```
enum Column {
    Col1(String),
    Col2(String),
    Col3(String),
}
```

Next, I chose to create a closure to handle the printing of the output. This also means I must figure out what goes in the first two columns based on the configuration:

```
    let default_col1 = if config.suppress_col1 { ❶
        ""
    } else {
        &config.delimiter
    };

    let default_col2 = if config.suppress_col2 { ❷
        ""
    } else {
        &config.delimiter
    };

    let printer = |col: Column| {
        let out = match col {
            Col1(val) => { ❸
                if config.suppress_col1 {
                    "".to_string()
                } else {
                    val
                }
            }
            Col2(val) => format!( ❹
                "{}{}",
                default_col1,
                if config.suppress_col2 { "" } else { &val },
            ),
            Col3(val) => format!( ❺
                "{}{}{}",
                default_col1,
                default_col2,
```

```
                if config.suppress_col3 { "" } else { &val },
            ),
        };

        if !out.trim().is_empty() { ❻
            println!("{}", out);
        }
    };
```

❶ When suppressing column 1, use the empty string; otherwise use the output delimiter.

❷ Do the same for column 2.

❸ Given the text for column 1, decide whether to print or suppress the output.

❹ Given the text for column 2, use the default value for column 1 and either print or suppress this column.

❺ Given the text for column 3, use the default values for columns 1 and 2 and either print or suppress this column.

❻ Only print nonempty lines of output.

Here is how I can use this idea:

```
let mut line1 = lines1.next(); ❶
let mut line2 = lines2.next();

loop {
    match (&line1, &line2) {
        (Some(val1), Some(val2)) => match val1.cmp(val2) {
            Equal => {
                printer(Col3(val1.to_string())); ❷
                line1 = lines1.next();
                line2 = lines2.next();
            }
            Less => {
                printer(Col1(val1.to_string())); ❸
                line1 = lines1.next();
            }
            _ => {
                printer(Col2(val2.to_string())); ❹
                line2 = lines2.next();
```

```
                }
            },
            (Some(val1), None) => {
                printer(Col1(val1.to_string())); ❺
                line1 = lines1.next();
            }
            (None, Some(val2)) => {
                printer(Col2(val2.to_string())); ❻
                line2 = lines2.next();
            }
            (None, None) => break,
        };
    }
```

❶ Draw the initial values from the two input files.

❷ When the values are the same, print one of them in column 3.

❸ When the first value is less than the second, print the first value in column 1.

❹ Otherwise, print the second value in column 2.

❺ When there is only a value from the first file, print it in column 1.

❻ When there is only a value from the second file, print it in column 2.

I like having the option to change the output delimiter from a tab to something more visible:

```
$ cargo run -- -d="--->" tests/inputs/file1.txt
tests/inputs/file2.txt
--->B
a
b
--->--->c
d
```

With these changes, all the tests pass.

Going Further

- Alter the program to show the GNU output and add those options.

- As I noted in the chapter introduction, `comm` performs basic join operations on two files which is similar to the `join` program. Read the manual page for that program and use your experience from writing `commr` to write `joinr`.

Summary

Like I said, `comm` was always black magic to me, and I had to look up the manual page every time to remember what the flags meant. Now that I've written my own version, I understand it much better and quite love the elegance of how it works. Consider what you learned:

- You can manually iterate the lines of a filehandle by calling `Iterator::next` on `BufRead::lines`.
- It's possible `match` on combinations of possibilities by grouping them into a tuple.
- You can use `std::cmp` to compare one value to another. The result is a variant of `std::cmp::Ordering`.
- You create an `enum` called `Column` where the variants can hold a string value like `Col1(String)`, which is really handy.

Chapter 11. Epilogue

No one in the world ever gets what they want and that is beautiful.
Everybody dies frustrated and sad and that is beautiful.

—They Might Be Giants

The end.

About the Author

Ken Youens-Clark has been working as a professional software developer for 25 years. Initially a student of Jazz Studies (drums) in undergrad at the University of North Texas, he changed major several times before limping out of school with a BA in English literature. Ken learned coding on the job starting in the mid-1990s and has worked in industry, research, and nonprofits over the years. In 2019, he earned his MS in Biosystems Engineering from the University of Arizona. He has previously published two books, Tiny Python Projects (Manning, 2020) and Mastering Python for Bioinformatics (O'Reilly, 2021). He resides in Tucson, Arizona, USA, with his wife, three children, and dog.