[Accepting Command Line Arguments 2](#_Toc496541041)

[Reading the Argument Values 3](#_Toc496541042)

[Saving the Argument Values in Variables 5](#_Toc496541043)

[Reading a File 6](#_Toc496541044)

[Refactoring to Improve Modularity and Error Handling 8](#_Toc496541045)

[Separation of Concerns for Binary Projects 9](#_Toc496541046)

[Extracting the Argument Parser 10](#_Toc496541047)

[Grouping Configuration Values 11](#_Toc496541048)

[Creating a Constructor for Config 13](#_Toc496541049)

[Fixing the Error Handling 14](#_Toc496541050)

[Improving the Error Message 14](#_Toc496541051)

[Returning a Result from new Instead of Calling panic! 15](#_Toc496541052)

[Calling Config::new and Handling Errors 16](#_Toc496541053)

[Extracting Logic from main 17](#_Toc496541054)

[Returning Errors from the run Function 18](#_Toc496541055)

[Handling Errors Returned from run in main 20](#_Toc496541056)

[Splitting Code into a Library Crate 21](#_Toc496541057)

[Developing the Library’s Functionality with Test Driven Development 23](#_Toc496541058)

[Writing a Failing Test 23](#_Toc496541059)

[Writing Code to Pass the Test 26](#_Toc496541060)

[Iterating Through Lines with the lines Method 26](#_Toc496541061)

[Searching Each Line for the Query 27](#_Toc496541062)

[Storing Matching Lines 27](#_Toc496541063)

[Using the search Function in the run Function 28](#_Toc496541064)

[Working with Environment Variables 30](#_Toc496541065)

[Writing a Failing Test for the Case-Insensitive search Function 30](#_Toc496541066)

[Implementing the search\_case\_insensitive Function 31](#_Toc496541067)

[Writing Error Messages to Standard Error Instead of Standard Output 36](#_Toc496541068)

[Checking Where Errors are Written to 36](#_Toc496541069)

[Printing Errors to Standard Error 37](#_Toc496541070)

[Summary 38](#_Toc496541071)

CE: Please leave lower-case words in headings as lower-case, these are functions or variable names

12

An I/O Project: Building a Command Line Program

This chapter is a recap of the many skills you’ve learned so far and an exploration of a few more standard library features. We’ll build a command line tool that interacts with file and command line input/output to practice some of the Rust concepts you now have under your belt.

Rust’s speed, safety, single binary output, and cross-platform support make it an ideal language for creating command line tools, so for our project, we’ll make our own version of the classic command line tool grep (globally search a regular expression and print). In the simplest use case, grep searches a specified file for a specified string. To do so, grep takes as its arguments a filename and a string, and then reads the file and finds lines in that file that contain the string argument. It then prints those lines.

Along the way, we’ll show how to make our command line tool use features of the terminal that many command line tools use. We’ll read the value of an environment variable to allow the user to configure the behavior of our tool. We’ll also print to the standard error console stream (stderr) instead of standard output (stdout), so, for example, the user can redirect successful output to a file while still seeing error messages onscreen.

One Rust community member, Andrew Gallant, has already created a fully featured, very fast version of grep, called ripgrep. By comparison, our version of grep will be fairly simple, but this chapter will give you some of the background knowledge you need to understand a real-world project like ripgrep.

Our grep project will combine a number of concepts you’ve learned so far:

Organizing code (using what you learned in modules, Chapter 7)

Using vectors and strings (collections, Chapter 8)

Handling errors (Chapter 9)

Using traits and lifetimes where appropriate (Chapter 10)

Writing tests (Chapter 11)

We’ll also briefly introduce closures, iterators, and trait objects, which Chapters 13 and 17 will cover in detail.

prod: confrim xrefs

Accepting Command Line Arguments

Let’s create a new project with, as always, cargo new. We’ll call our project minigrep to distinguish it from the grep tool that you might already have on your system.

$ cargo new --bin minigrep

Created binary (application) `minigrep` project

$ cd minigrep

The first task is to make minigrep accept its two command line arguments: the filename and a string to search for. That is, we want to be able to run our program with cargo run, a string to search for, and a path to a file to search in, like so:

$ cargo run searchstring example-filename.txt

Right now, the program generated by cargo new cannot process arguments we give it. However, some existing libraries on https://crates.io/ can help us accept command line arguments, but because you’re just learning this concept, let’s implement this capability ourselves.

Prod: s/b “https://crates.io/” in ital because it’s a URL. Update globally

Reading the Argument Values

To make sure minigrep is able to get the values of command line arguments we pass to it, we’ll need a function provided in Rust’s standard library, which is std::env::args. This function returns an iterator of the command line arguments that were given to minigrep. We haven’t discussed iterators yet (we’ll cover them fully in Chapter 13), but for now you only need to know two details about iterators: iterators produce a series of values, and we can call the collect function on an iterator to turn it into a collection, such as a vector, containing all the elements the iterator produces.

Prod: Confrim xref

Use the code in Listing 12-1 to allow your minigrep program to read any command line arguments passed to it and then collect the values into a vector.

src/main.rs

use std::env;

fn main() {

let args: Vec<String> = env::args().collect();

println!("{:?}", args);

}

Listing 12-1: Collecting the command line arguments into a vector and printing them

First, we bring the std::env module into scope with a use statement so we can use its args function. Notice that the std::env::args function is nested in two levels of modules. As we discussed in Chapter 7, in cases where the desired function is nested in more than one module, it’s conventional to bring the parent module into scope rather than the function. As a result, we can easily use other functions from std::env. It’s also less ambiguous than adding use std::env::args and then calling the function with just args because args might easily be mistaken for a function that’s defined in the current module.

prod: confirm xref

Start box

The args Function and Invalid Unicode

Note that std::env::args will panic if any argument contains invalid Unicode. If your program needs to accept arguments containing invalid Unicode, use std::env::args\_os instead. That function returns OsString values instead of String values. We’ve chosen to use std::env::args here for simplicity because OsString values differ per platform and are more complex to work with than String values.

End box

On the first line of main, we call env::args, and immediately use collect to turn the iterator into a vector containing all the values produced by the iterator. We can use the collect function to create many kinds of collections, so we explicitly annotate the type of args to specify that we want a vector of strings. Although we very rarely need to annotate types in Rust, collect is one function you do often need to annotate because Rust isn’t able to infer the kind of collection you want.

Finally, we print the vector using the debug formatter, :?. Let’s try running the code with no arguments, and then with two arguments:

$ cargo run

["target/debug/minigrep"]

$ cargo run needle haystack

--snip--

["target/debug/minigrep", "needle", "haystack"]

Notice that the first value in the vector is "target/debug/minigrep", which is the name of our binary. This matches the behavior of the arguments list in C, letting programs use the name by which they were invoked in their execution. It’s often convenient to have access to the program name in case we want to print it in messages or change behavior of the program based on what command line alias was used to invoke the program. But for the purposes of this chapter, we’ll ignore it and save only the two arguments we need.

Saving the Argument Values in Variables

Printing the value of the vector of arguments illustrated that the program is able to access the values specified as command line arguments. Now we need to save the values of the two arguments in variables so we can use the values throughout the rest of the program. We do that in Listing 12-2.

src/main.rs

use std::env;

fn main() {

let args: Vec<String> = env::args().collect();

let query = &args[1];

let filename = &args[2];

println!("Searching for {}", query);

println!("In file {}", filename);

}

Listing 12-2: Creating variables to hold the query argument and filename argument

As we saw when we printed the vector, the program’s name takes up the first value in the vector at args[0], so we’re starting at index 1. The first argument minigrep takes is the string we’re searching for, so we put a reference to the first argument in the variable query. The second argument will be the filename, so we put a reference to the second argument in the variable filename.

We temporarily print the values of these variables to prove that the code is working as we intend. Let’s run this program again with the arguments test and sample.txt:

$ cargo run test sample.txt

Finished dev [unoptimized + debuginfo] target(s) in 0.0 secs

Running `target/debug/minigrep test sample.txt`

Searching for test

In file sample.txt

Great, the program is working! The values of the arguments we need are being saved into the right variables. Later we’ll add some error handling to deal with certain potential erroneous situations, such as when the user provides no arguments; for now, we’ll ignore that situation and work on adding file reading capabilities instead.

Reading a File

Now we’ll add functionality to read the file that is specified in the filename command line argument. First, we need a sample file to test it with: the best kind of file to use to make sure minigrep is working is one with a small amount of text over multiple lines with some repeated words. Listing 12-3 has an Emily Dickinson poem that will work well! Create a file called poem.txt at the root level of your project, and enter the poem “I’m Nobody! Who are you?”

poem.txt

I’m nobody! Who are you?

Are you nobody, too?

Then there’s a pair of us — don’t tell!

They’d banish us, you know.

How dreary to be somebody!

How public, like a frog

To tell your name the livelong day

To an admiring bog!

Listing 12-3: A poem by Emily Dickinson will make a good test case.

With the text in place, edit src/main.rs and add code to open the file, as shown in Listing 12-4.

src/main.rs

use std::env;

use std::fs::File;

use std::io::prelude::\*;

fn main() {

// --snip--

println!("In file {}", filename);

let mut f = File::open(filename).expect("file not found");

let mut contents = String::new();

f.read\_to\_string(&mut contents)

.expect("something went wrong reading the file");

println!("With text:\n{}", contents);

}

Listing 12-4: Reading the contents of the file specified by the second argument

First, we add some more use statements to bring in relevant parts of the standard library: we need std::fs::File to handle files, and std::io::prelude::\* contains various useful traits for doing I/O, including file I/O. In the same way that Rust has a general prelude that brings certain things into scope automatically, the std::io module has its own prelude of common things you’ll need when working with I/O. Unlike the default prelude, we must explicitly use the prelude from std::io.

In main, we’ve added three items: first, we get a mutable handle to the file by calling the File::open function and passing it the value of the filename variable. Second, we create a variable called contents and set it to a mutable, empty String. This will hold the content of the file after we read it in. Third, we call read\_to\_string on our file handle and pass a mutable reference to contents as an argument.

After those lines, we’ve again added a temporary println! statement that prints the value of contents after the file is read, so we can check that the program is working so far.

Let’s run this code with any string as the first command line argument (because we haven’t implemented the searching part yet) and the poem.txt file as the second argument:

$ cargo run the poem.txt

Finished dev [unoptimized + debuginfo] target(s) in 0.0 secs

Running `target/debug/minigrep the poem.txt`

Searching for the

In file poem.txt

With text:

I’m nobody! Who are you?

Are you nobody, too?

Then there’s a pair of us — don’t tell!

They’d banish us, you know.

How dreary to be somebody!

How public, like a frog

To tell your name the livelong day

To an admiring bog!

Great! The code read and then printed the content of the file. But a few flaws are visible. The main function has multiple responsibilities: generally, functions are clearer and easier to maintain if each function is responsible for only one idea. The other problem is that we’re not handling errors as well as we could be. Although the program is still small, these flaws aren’t a big problem, but as the program grows, it will be harder to fix them cleanly. It’s good practice to begin refactoring early on when developing a program, because it’s much easier to refactor smaller amounts of code. We’ll do that next.

Refactoring to Improve Modularity and Error Handling

To improve our program, we’ll fix four problems that have to do with the program’s structure and how it’s handling potential errors.

First, our main function now performs two tasks: it parses arguments and opens files. For such a small function, this isn’t a major problem. However, if we continue to grow our program inside main, the number of separate tasks the main function handles will increase. As a function gains responsibilities, it becomes more difficult to reason about, harder to test, and harder to change without breaking one of its parts. It’s best to separate functionality so each function is responsible for one task.

This issue also ties into the second problem: although query and filename are configuration variables to our program, variables like f and contents are used to perform the program’s logic. The longer main becomes, the more variables we’ll need to bring into scope; the more variables we have in scope, the harder it will be to keep track of the purpose of each. It’s best to group the configuration variables into one structure to make their purpose clear.

The third problem is that we’ve used expect to print an error message when opening the file fails, but the error message just prints file not found. Opening a file can fail in a number of ways besides the file being missing: for example, the file might exist, but we might not have permission to open it. Right now, if we’re in that situation, we’d print the file not found error message that would give the user the wrong information!

Fourth, we use expect repeatedly to handle different errors, and if the user runs our program without specifying enough arguments, they’ll get an index out of bounds error from Rust that doesn’t clearly explain the problem. It would be best if all the error handling code was in one place so future maintainers have only one place to consult in the code if the error handling logic needs to change. Having all the error handling code in one place will also ensure that we’re printing messages that will be meaningful to our end users.

Let’s address these four problems by refactoring our project.

Separation of Concerns for Binary Projects

The organizational problem of allocating responsibility for multiple tasks to the main function is common to many binary projects. As a result, the Rust community has developed a type of guideline process for splitting the separate concerns of a binary program when main starts getting large. The process has the following steps:

Split your program into a main.rs and a lib.rs, and move your program’s logic to lib.rs.

While your command line parsing logic is small, it can remain in main.rs.

When the command line parsing logic starts getting complicated, extract it from main.rs and move it to lib.rs.

The responsibilities that remain in the main function after this process should be limited to:

Calling the command line parsing logic with the argument values

Setting up any other configuration

Calling a run function in lib.rs

Handling the error if run returns an error

This pattern is about separating concerns: main.rs handles running the program, and lib.rs handles all the logic of the task at hand. Because we can’t test the main function directly, this structure lets us test all of our program’s logic by moving it into functions in lib.rs. The only code that remains in main.rs will be small enough to verify its correctness by reading it. Let’s rework our program by following this process.

Extracting the Argument Parser

We’ll extract the functionality for parsing arguments into a function that main will call to prepare for moving the command line parsing logic to src/lib.rs. Listing 12-5 shows the new start of main that calls a new function parse\_config, which we’ll define in src/main.rs for the moment.

src/main.rs

fn main() {

let args: Vec<String> = env::args().collect();

let (query, filename) = parse\_config(&args);

// --snip--

}

fn parse\_config(args: &[String]) -> (&str, &str) {

let query = &args[1];

let filename = &args[2];

(query, filename)

}

Listing 12-5: Extracting a parse\_config function from main

We’re still collecting the command line arguments into a vector, but instead of assigning the argument value at index 1 to the variable query and the argument value at index 2 to the variable filename within the main function, we pass the whole vector to the parse\_config function. The parse\_config function then holds the logic that determines which argument goes in which variable and passes the values back to main. We still create the query and filename variables in main, but main no longer has the responsibility of determining how the command line arguments and variables correspond.

This rework may seem like overkill for our small program, but we’re refactoring in small, incremental steps. After making this change, run the program again to verify that the argument parsing still works. It’s good to check your progress often, because that will help you identify the cause of problems when they occur.

Grouping Configuration Values

We can take another small step to improve this function further. At the moment, we’re returning a tuple, but then we immediately break that tuple into individual parts again. This is a sign that perhaps we don’t have the right abstraction yet.

Another indicator that shows there’s room for improvement is the config part of parse\_config, which implies that the two values we return are related and are both part of one configuration value. We’re not currently conveying this meaning in the structure of the data other than grouping the two values into a tuple: we could put the two values into one struct and give each of the struct fields a meaningful name. Doing so will make it easier for future maintainers of this code to understand how the different values relate to each other and what their purpose is.

Note Some people call this anti-pattern of using primitive values when a complex type would be more appropriate primitive obsession.

Listing 12-6 shows the addition of a struct named Config defined to have fields named query and filename. We’ve also changed the parse\_config function to return an instance of the Config struct and updated main to use the struct fields rather than having separate variables.

src/main.rs

fn main() {

let args: Vec<String> = env::args().collect();

let config = parse\_config(&args);

println!("Searching for {}", config.query);

println!("In file {}", config.filename);

let mut f = File::open(config.filename).expect("file not found");

// --snip--

}

struct Config {

query: String,

filename: String,

}

fn parse\_config(args: &[String]) -> Config {

let query = args[1].clone();

let filename = args[2].clone();

Config { query, filename }

}

Listing 12-6: Refactoring parse\_config to return an instance of a Config struct

The signature of parse\_config now indicates that it returns a Config value. In the body of parse\_config, where we used to return string slices that reference String values in args, we now define Config to contain owned String values. The args variable in main is the owner of the argument values and is only letting the parse\_config function borrow them, which means we’d violate Rust’s borrowing rules if Config tried to take ownership of the values in args.

We could manage the String data in a number of different ways, but the easiest, though somewhat inefficient, route is to call the clone method on the values. This will make a full copy of the data for the Config instance to own, which takes more time and memory than storing a reference to the string data. However, cloning the data also makes our code very straightforward because we don’t have to manage the lifetimes of the references; in this circumstance, giving up a little performance to gain simplicity is a worthwhile trade-off.

Start box

The Trade-Offs of Using clone

There’s a tendency among many Rustaceans to avoid using clone to fix ownership problems because of its runtime cost. In Chapter 13, you’ll learn how to use more efficient methods in this type of situation. But for now, it’s okay to copy a few strings to continue making progress because we’ll make these copies only once, and our filename and query string are very small. It’s better to have a working program that is a bit inefficient than to try to hyperoptimize code on your first pass. As you become more experienced with Rust, it will be easier to use the desirable method, but for now, it’s perfectly acceptable to call clone.

End box

prod: confirm xref

We’ve updated main so it places the instance of Config returned by parse\_config into a variable named config, and we updated the code that previously used the separate query and filename variables so it now uses the fields on the Config struct instead.

Now our code more clearly conveys that query and filename are related, and their purpose is to configure how the program will work. Any code that uses these values knows to find them in the config instance in the fields named for their purpose.

Creating a Constructor for Config

So far, we’ve extracted the logic responsible for parsing the command line arguments from main into the parse\_config function, which helped us to see that the query and filename values were related and that relationship should be conveyed in our code. We then added a Config struct to name the related purpose of query and filename, and to be able to return the values’ names as struct field names from the parse\_config function.

So now that the purpose of the parse\_config function is to create a Config instance, we can change parse\_config from being a plain function to a function named new that is associated with the Config struct. Making this change will make the code more idiomatic: we can create instances of types in the standard library, such as String, by calling String::new, and by changing parse\_config into a new function associated with Config, we’ll be able to create instances of Config by calling Config::new. Listing 12-7 shows the changes we need to make.

src/main.rs

fn main() {

let args: Vec<String> = env::args().collect();

let config = Config::new(&args);

// --snip--

}

// --snip--

impl Config {

fn new(args: &[String]) -> Config {

let query = args[1].clone();

let filename = args[2].clone();

Config { query, filename }

}

}

Listing 12-7: Changing parse\_config into Config::new

We’ve updated main where we were calling parse\_config to instead call Config::new. We’ve changed the name of parse\_config to new and moved it within an impl block, which associates the new function with Config. Try compiling this code again to make sure it works.

Fixing the Error Handling

Now we’ll work on fixing our error handling. Recall that attempting to access the values in the args vector at index 1 or index 2 will cause the program to panic if the vector contains fewer than three items. Try running the program without any arguments: it will look like this:

$ cargo run

Finished dev [unoptimized + debuginfo] target(s) in 0.0 secs

Running `target/debug/minigrep`

thread 'main' panicked at 'index out of bounds: the len is 1

but the index is 1', /stable-dist-rustc/build/src/libcollections/vec.rs:1307

note: Run with `RUST\_BACKTRACE=1` for a backtrace.

The line index out of bounds: the len is 1 but the index is 1 is an error message intended for programmers. It won’t help our end users understand what happened and what they should do instead. Let’s fix that now.

Improving the Error Message

In Listing 12-8, we add a check in the new function that will verify that the slice is long enough before accessing index 1 and 2. If the slice isn’t long enough, the program panics and displays a better error message than the index out of bounds message.

src/main.rs

// --snip--

fn new(args: &[String]) -> Config {

if args.len() < 3 {

panic!("not enough arguments");

}

// --snip--

Listing 12-8: Adding a check for the number of arguments

This code is similar to the Guess::new function we wrote in Listing 9-8 where we called panic! when the value argument was out of the range of valid values. Instead of checking for a range of values here, we’re checking that the length of args is at least 3 and the rest of the function can operate under the assumption that this condition has been met. If args has fewer than three items, this condition will be true, and we call the panic! macro to end the program immediately.

With these extra few lines of code in new, let’s run the program without any arguments again to see what the error looks like now:

$ cargo run

Finished dev [unoptimized + debuginfo] target(s) in 0.0 secs

Running `target/debug/minigrep`

thread 'main' panicked at 'not enough arguments', src/main.rs:29

note: Run with `RUST\_BACKTRACE=1` for a backtrace.

This output is better: we now have a reasonable error message. However, we also have extraneous information we don’t want to give to our users. Perhaps using the technique we used in Listing 9-8 isn’t the best to use here: a call to panic! is more appropriate for a programming problem rather than a usage problem, as discussed in Chapter 9. Instead, we can use the other technique you learned about in Chapter 9—returning a Result that indicates either success or an error.

Prod: confrim xrefs

Returning a Result from new Instead of Calling panic!

We can instead return a Result value that will contain a Config instance in the successful case and will describe the problem in the error case. When Config::new is communicating to main, we can use the Result type to signal there was a problem. Then we can change main to convert an Err variant into a more practical error for our users without the surrounding text about thread 'main' and RUST\_BACKTRACE that a call to panic! causes.

Listing 12-9 shows the changes we need to make to the return value of Config::new and the body of the function needed to return a Result.

src/main.rs

impl Config {

fn new(args: &[String]) -> Result<Config, &'static str> {

if args.len() < 3 {

return Err("not enough arguments");

}

let query = args[1].clone();

let filename = args[2].clone();

Ok(Config { query, filename })

}

}

Listing 12-9: Returning a Result from Config::new

Our new function now returns a Result with a Config instance in the success case and a &'static str in the error case. Recall from “The Static Lifetime” section in Chapter 10 that &'static str is the type of string literals, which is our error message type for now.

prod: confrim xref

We’ve made two changes in the body of the new function: instead of calling panic! when the user doesn’t pass enough arguments, we now return an Err value, and we’ve wrapped the Config return value in an Ok. These changes make the function conform to its new type signature.

Returning an Err value from Config::new allows the main function to handle the Result value returned from the new function and exit the process more cleanly in the error case.

Calling Config::new and Handling Errors

To handle the error case and print a user-friendly message, we need to update main to handle the Result being returned by Config::new, as shown in Listing 12-10. We’ll also take the responsibility of exiting the command line tool with a nonzero error code from panic! and implement it by hand. A nonzero exit status is a convention to signal to the process that called our program that the program exited with an error state.

src/main.rs

use std::process;

fn main() {

let args: Vec<String> = env::args().collect();

let config = Config::new(&args).unwrap\_or\_else(|err| {

println!("Problem parsing arguments: {}", err);

process::exit(1);

});

// --snip--

Listing 12-10: Exiting with an error code if creating a new Config fails

In this listing, we’ve used a method we haven’t covered before: unwrap\_or\_else, which is defined on Result<T, E> by the standard library. Using unwrap\_or\_else allows us to define some custom, non-panic! error handling. If the Result is an Ok value, this method’s behavior is similar to unwrap: it returns the inner value Ok is wrapping. However, if the value is an Err value, this method calls the code in the closure, which is an anonymous function we define and pass as an argument to unwrap\_or\_else. We’ll cover closures in more detail in Chapter 13. For now, you just need to know that unwrap\_or\_else will pass the inner value of the Err, which in this case is the static string not enough arguments that we added in Listing 12-9, to our closure in the argument err that appears between the vertical pipes. The code in the closure can then use the err value when it runs.

Prod: Confrim xref

We’ve added a new use line to import process from the standard library. The code in the closure that will be run in the error case is only two lines: we print the err value and then call process::exit. The process::exit function will stop the program immediately and return the number that was passed as the exit status code. This is similar to the panic!-based handling we used in Listing 12-8, but we no longer get all the extra output. Let’s try it:

$ cargo run

Compiling minigrep v0.1.0 (file:///projects/minigrep)

Finished dev [unoptimized + debuginfo] target(s) in 0.48 secs

Running `target/debug/minigrep`

Problem parsing arguments: not enough arguments

Great! This output is much friendlier for our users.

Extracting Logic from main

Now that we’ve finished refactoring the configuration parsing, let’s turn to the program’s logic. As we stated in “Separation of Concerns for Binary Projects” on page XX, we’ll extract a function named run that will hold all the logic currently in the main function that is not involved with setting up configuration or handling errors. When we’re done, main will be concise and easy to verify by inspection, and we’ll be able to write tests for all the other logic.

Prod: fill/confirm xref

Listing 12-11 shows the extracted run function. For now, we’re just making the small, incremental improvement of extracting the function. We’re still defining the function in src/main.rs.

Prod: See the cross-reference above.

src/main.rs

fn main() {

// --snip--

println!("Searching for {}", config.query);

println!("In file {}", config.filename);

run(config);

}

fn run(config: Config) {

let mut f = File::open(config.filename).expect("file not found");

let mut contents = String::new();

f.read\_to\_string(&mut contents)

.expect("something went wrong reading the file");

println!("With text:\n{}", contents);

}

// --snip--

Listing 12-11: Extracting a run function containing the rest of the program logic

The run function now contains all the remaining logic from main, starting from reading the file. The run function takes the Config instance as an argument.

Returning Errors from the run Function

With the remaining program logic separated into the run function, we can improve the error handling, as we did with Config::new in Listing 12-9. Instead of allowing the program to panic by calling expect, the run function will return a Result<T, E> when something goes wrong. This will let us further consolidate into main the logic around handling errors in a user-friendly way. Listing 12-12 shows the changes we need to make to the signature and body of run.

src/main.rs

use std::error::Error;

// --snip--

fn run(config: Config) -> Result<(), Box<Error>> {

let mut f = File::open(config.filename)?;

let mut contents = String::new();

f.read\_to\_string(&mut contents)?;

println!("With text:\n{}", contents);

Ok(())

}

Listing 12-12: Changing the run function to return Result

We’ve made three significant changes here. First, we changed the return type of the run function to Result<(), Box<Error>>. This function previously returned the unit type, (), and we keep that as the value returned in the Ok case.

For the error type, we used the trait object Box<Error> (and we’ve brought std::error::Error into scope with a use statement at the top). We’ll cover trait objects in Chapter 17. For now, just know that Box<Error> means the function will return a type that implements the Error trait, but we don’t have to specify what particular type the return value will be. This gives us flexibility to return error values that may be of different types in different error cases.

Prod: confirm xref

Second, we’ve removed the calls to expect in favor of ?, as we talked about in Chapter 9. Rather than panic! on an error, this change to ? will return the error value from the current function for the caller to handle.

prod: confrim xref

Third, this function now returns an Ok value in the success case. We’ve declared the run function’s success type as () in the signature, which means we need to wrap the unit type value in the Ok value. This Ok(()) syntax might look a bit strange at first, but using () like this is the idiomatic way to indicate that we’re calling run for its side effects only; it doesn’t return a value we need.

When you run this code, it will compile but will display a warning:

warning: unused result which must be used, #[warn(unused\_must\_use)] on by default

--> src/main.rs:39:5

|

39 | run(config);

| ^^^^^^^^^^^^

Rust tells us that our code ignored the Result value, which might indicate that an error occurred. But we’re not checking to see whether or not there was an error, and the compiler reminds us that we probably meant to have some error handling code here! Let’s rectify that problem now.

Handling Errors Returned from run in main

We’ll check for errors and handle them using a technique similar to the way we handled errors with Config::new in Listing 12-10, but with a slight difference.

src/main.rs

fn main() {

// --snip--

println!("Searching for {}", config.query);

println!("In file {}", config.filename);

if let Err(e) = run(config) {

println!("Application error: {}", e);

process::exit(1);

}

}

We use if let rather than unwrap\_or\_else to check whether run returns an Err value and call process::exit(1) if it does. The run function doesn’t return a value that we want to unwrap in the same way that Config::new returns the Config instance. Because run returns a () in the success case, we only care about detecting an error, so we don’t need unwrap\_or\_else to return the unwrapped value because it would only be ().

The bodies of the if let and the unwrap\_or\_else functions are the same in both cases: we print the error and exit.

Splitting Code into a Library Crate

Our minigrep project is looking good so far! Now we’ll split the src/main.rs file and put some code into the src/lib.rs file so we can test it and have a src/main.rs file with fewer responsibilities.

Let’s move all the code that isn’t the main function from src/main.rs to src/lib.rs:

The run function definition

The relevant use statements

The definition of Config

The Config::new function definition

The contents of src/lib.rs should have the signatures shown in Listing 12-13 (we’ve omitted the bodies of the functions for brevity).

src/lib.rs

use std::error::Error;

use std::fs::File;

use std::io::prelude::\*;

pub struct Config {

pub query: String,

pub filename: String,

}

impl Config {

pub fn new(args: &[String]) -> Result<Config, &'static str> {

// --snip--

}

}

pub fn run(config: Config) -> Result<(), Box<Error>> {

// --snip--

}

Listing 12-13: Moving Config and run into src/lib.rs

We’ve made liberal use of pub here: on Config, its fields and its new method, and on the run function. We now have a library crate that has a public API that we can test!

Now we need to bring the code we moved to src/lib.rs into the scope of the binary crate in src/main.rs by using extern crate minigrep. Then we’ll add a use minigrep::Config line to bring the Config type into scope and prefix the run function with our crate name, as shown in Listing 12-14.

src/main.rs

extern crate minigrep;

use std::env;

use std::process;

use minigrep::Config;

fn main() {

// --snip--

if let Err(e) = minigrep::run(config) {

// --snip--

}

}

Listing 12-14: Bringing the minigrep crate into the scope of src/main.rs

To bring the library crate into the binary crate, we use extern crate minigrep. Then we’ll add a use minigrep::Config line to bring the Config type into scope, and we’ll prefix the run function with our crate name. Now, all the functionality should be connected and should work. Run the program with cargo run and make sure everything works correctly.

Whew! That was a lot of work, but we’ve set ourselves up for success in the future. Now it’s much easier to handle errors, and we’ve made the code more modular. Almost all of our work will be done in src/lib.rs from here on out.

Let’s take advantage of this newfound modularity by doing something that would have been difficult with the old code but is easy with our new code: we’ll write some tests!

Developing the Library’s Functionality with Test Driven Development

Now that we’ve extracted the logic into src/lib.rs and left the argument collecting and error handling in src/main.rs, it’s much easier to write tests for the core functionality of our code. We can call functions directly with various arguments and check return values without having to call our binary from the command line. Feel free to write some tests for the functionality in the Config::new and run functions on your own.

In this section, we’ll add the minigrep program’s searching logic by using the Test Driven Development (TDD) process. This software development technique follows these steps:

Write a test that fails, and run it to make sure it fails for the reason you expected.

Write or modify just enough code to make the new test pass.

Refactor the code you just added or changed, and make sure the tests continue to pass.

Repeat from step 1!

This process is just one of many ways to write software, but TDD can help drive code design as well. Writing the test before you write the code that makes the test pass helps to maintain high test coverage throughout the process.

We’ll test drive the implementation of the functionality that will actually do the searching for the query string in the file contents and produce a list of lines that match the query. We’ll add this functionality in a function called search.

Writing a Failing Test

Because we don’t need them any more, let’s remove the println! statements from src/lib.rs and src/main.rs. Then we’ll add a test module with a test function, as we did in Chapter 11. The test function specifies the behavior we want the search function to have: it will take a query and the text to search for the query in, and will return only the lines from the text that contain the query. Listing 12-15 shows this test.

Prod: confirm xref

src/lib.rs

#[cfg(test)]

mod test {

use super::\*;

#[test]

fn one\_result() {

let query = "duct";

let contents = "\

Rust:

safe, fast, productive.

Pick three.";

assert\_eq!(

vec!["safe, fast, productive."],

search(query, contents)

);

}

}

Listing 12-15: Creating a failing test for the search function we wish we had

This test searches for the string “duct.” The text we’re searching is three lines, only one of which contains “duct.” We assert that the value returned from the search function contains only the line we expect.

We aren’t able to run this test and watch it fail because the test doesn’t even compile: the search function doesn’t exist yet! So now we’ll add just enough code to get the test to compile and run by adding a definition of the search function that always returns an empty vector, as shown in Listing 12-16. Then the test should compile and fail because an empty vector doesn’t match a vector containing the line "safe, fast, productive.".

src/lib.rs

pub fn search<'a>(query: &str, contents: &'a str) -> Vec<&'a str> {

vec![]

}

Listing 12-16: Defining just enough of the search function so our test will compile

Notice that we need an explicit lifetime 'a defined in the signature of search and used with the contents argument and the return value. Recall in Chapter 10 that the lifetime parameters specify which argument lifetime is connected to the lifetime of the return value. In this case, we indicate that the returned vector should contain string slices that reference slices of the argument contents (rather than the argument query).

Prod: Confrim xref

In other words, we tell Rust that the data returned by the search function will live as long as the data passed into the search function in the contents argument. This is important! The data referenced by a slice needs to be valid for the reference to be valid; if the compiler assumes we’re making string slices of query rather than contents, it will do its safety checking incorrectly.

When we try to compile this function without lifetimes, we’ll get this error:

error[E0106]: missing lifetime specifier

--> src/lib.rs:5:47

|

5 | fn search(query: &str, contents: &str) -> Vec<&str> {

| ^ expected lifetime parameter

|

= help: this function's return type contains a borrowed value, but the

signature does not say whether it is borrowed from `query` or `contents`

Rust can’t possibly know which of the two arguments we need, so we need to tell it. Because contents is the argument that contains all of our text and we want to return the parts of that text that match, we know contents is the argument that should be connected to the return value using the lifetime syntax.

Other programming languages don’t require you to connect arguments to return values in the signature, so although this might seem strange, it will get easier over time. You might want to compare this example with “Lifetime Syntax” in Chapter 10 on page XX.

Now let’s run the test:

Prod: confirm/fill cross-reference above.

$ cargo test

...warnings...

Finished dev [unoptimized + debuginfo] target(s) in 0.43 secs

Running target/debug/deps/minigrep-abcabcabc

running 1 test

test test::one\_result ... FAILED

failures:

---- test::one\_result stdout ----

thread 'test::one\_result' panicked at 'assertion failed: `(left == right)`

(left: `["safe, fast, productive."]`, right: `[]`)', src/lib.rs:16

note: Run with `RUST\_BACKTRACE=1` for a backtrace.

failures:

test::one\_result

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

error: test failed

Great, the test fails, exactly as we expected. Let’s get the test to pass!

Writing Code to Pass the Test

Currently, our test is failing because we always return an empty vector. To fix that and implement search, our program needs to follow these steps:

Iterate through each line of the contents.

Check whether the line contains our query string.

If it does, add it to the list of values we’re returning.

If it doesn’t, do nothing.

Return the list of results that match.

Let’s work through each step, starting with iterating through lines.

Iterating Through Lines with the lines Method

Rust has a helpful method to handle line-by-line iteration of strings, conveniently named lines, that works as shown in Listing 12-17.

src/lib.rs

pub fn search<'a>(query: &str, contents: &'a str) -> Vec<&'a str> {

for line in contents.lines() {

// do something with line

}

}

Listing 12-17: Iterating through each line in contents

The lines method returns an iterator. We’ll talk about iterators in depth in Chapter 13, but recall that you saw this way of using an iterator in Listing 3-6 on page XX, where we used a for loop with an iterator to run some code on each item in a collection.

Prod: Confrim xref to Ch 13

Prod: Fill page xref for Listing 3-6

Searching Each Line for the Query

To check whether the current line contains our query string, we’ll add functionality. Fortunately, strings have a helpful method named contains that does this for us! Add a call to the contains method in the search function, as shown in Listing 12-18.

src/lib.rs

pub fn search<'a>(query: &str, contents: &'a str) -> Vec<&'a str> {

for line in contents.lines() {

if line.contains(query) {

// do something with line

}

}

}

Listing 12-18: Adding functionality to see whether the line contains the string in query

Storing Matching Lines

We also need a way to store the lines that contain our query string. For that, we can make a mutable vector before the for loop and call the push method to store a line in the vector. After the for loop, we return the vector, as shown in Listing 12-19.

src/lib.rs

pub fn search<'a>(query: &str, contents: &'a str) -> Vec<&'a str> {

let mut results = Vec::new();

for line in contents.lines() {

if line.contains(query) {

results.push(line);

}

}

results

}

Listing 12-19: Storing the lines that match so we can return them

Now the search function should return only the lines that contain query, and our test should pass. Let’s run the test:

$ cargo test

running 1 test

test test::one\_result ... ok

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

Our test passed, so we know it works!

At this point, we could consider opportunities for refactoring the implementation of the search function while keeping the code that passes the tests to maintain the same functionality. The code in the search function isn’t too bad, but it doesn’t take advantage of some useful features of iterators. We’ll return to this example in Chapter 13 where we’ll explore iterators in detail and look at how to improve it.Prod: confirm xref

Using the search Function in the run Function

Now that the search function is working and tested, we need to call search from our run function. We need to pass the config.query value and the contents that run reads from the file to the search function. Then run will print each line returned from search:

src/lib.rs

pub fn run(config: Config) -> Result<(), Box<Error>> {

let mut f = File::open(config.filename)?;

let mut contents = String::new();

f.read\_to\_string(&mut contents)?;

for line in search(&config.query, &contents) {

println!("{}", line);

}

Ok(())

}

We’re still using a for loop to return each line from search and print it.

Now the entire program should work! Let’s try it out, first with a word that should return exactly one line from the Emily Dickinson poem “frog”:

$ cargo run frog poem.txt

Compiling minigrep v0.1.0 (file:///projects/minigrep)

Finished dev [unoptimized + debuginfo] target(s) in 0.38 secs

Running `target/debug/minigrep frog poem.txt`

How public, like a frog

Cool! Now let’s try a word that will match multiple lines, like “the”:

$ cargo run the poem.txt

Finished dev [unoptimized + debuginfo] target(s) in 0.0 secs

Running `target/debug/minigrep the poem.txt`

Then there’s a pair of us — don’t tell!

To tell your name the livelong day

And finally, let’s make sure that we don’t get any lines when we search for a word that isn’t anywhere in the poem, such as “monomorphization”:

$ cargo run monomorphization poem.txt

Finished dev [unoptimized + debuginfo] target(s) in 0.0 secs

Running `target/debug/minigrep monomorphization poem.txt`

Excellent! We’ve built our own mini version of a classic tool and learned a lot about how to structure applications. We’ve also learned a bit about file input and output, lifetimes, testing, and command line parsing.

To round out this project chapter, we’ll briefly demonstrate how to work with environment variables and how to print to standard error, both of which are useful when you’re writing command line programs. Feel free to move on to Chapter 13 if you’d like at this point.

Working with Environment Variables

We’ll improve our tool by adding an extra feature: an option for case-insensitive searching that the user can turn on via an environment variable. We could make this feature a command line option and require that users enter it each time they want it to apply, but instead we’ll use an environment variable. Doing so allows our users to set the environment variable once and have all their searches be case insensitive in that terminal session.

Writing a Failing Test for the Case-Insensitive search Function

We want to add a new search\_case\_insensitive function that we’ll call when the environment variable is on. We’ll continue to follow the TDD process, so the first step is again to write a failing test. We’ll add a new test for the new case-insensitive search function and rename our old test from one\_result to case\_sensitive to clarify the differences between the two tests, as shown in Listing 12-20.

src/lib.rs

#[cfg(test)]

mod test {

use super::\*;

#[test]

fn case\_sensitive() {

let query = "duct";

let contents = "\

Rust:

safe, fast, productive.

Pick three.

Duct tape.";

assert\_eq!(

vec!["safe, fast, productive."],

search(query, contents)

);

}

#[test]

fn case\_insensitive() {

let query = "rUsT";

let contents = "\

Rust:

safe, fast, productive.

Pick three.

Trust me.";

assert\_eq!(

vec!["Rust:", "Trust me."],

search\_case\_insensitive(query, contents)

);

}

}

Listing 12-20: Adding a new failing test for the case-insensitive function we’re about to add

Note that we’ve edited the old test’s contents too. We’ve added a new line with the text “Duct tape” using a capital D that shouldn’t match the query “duct” when we’re searching in a case-sensitive manner. Changing the old test in this way helps ensure that we don’t accidentally break the case-sensitive search functionality that we’ve already implemented. This test should pass now and should continue to pass as we work on the case-insensitive search.

The new test for the case-insensitive search uses “rUsT” as its query. In the search\_case\_insensitive function we’re about to add, the query “rUsT” should match the line containing “Rust:” with a capital R and also the line “Trust me.” even though both have different casing than the query. This is our failing test, and it will fail to compile because we haven’t yet defined the search\_case\_insensitive function. Feel free to add a skeleton implementation that always returns an empty vector, similar to the way we did for the search function in Listing 12-16 to see the test compile and fail.

Implementing the search\_case\_insensitive Function

The search\_case\_insensitive function, shown in Listing 12-21, will be almost the same as the search function. The only difference is that we’ll lowercase the query and each line so whatever the case of the input arguments, they’ll be the same case when we check whether the line contains the query.

src/lib.rs

fn search\_case\_insensitive<'a>(query: &str, contents: &'a str) -> Vec<&'a str> {

let query = query.to\_lowercase();

let mut results = Vec::new();

for line in contents.lines() {

if line.to\_lowercase().contains(&query) {

results.push(line);

}

}

results

}

Listing 12-21: Defining the search\_case\_insensitive function to lowercase the query and the line before comparing them

First, we lowercase the query string and store it in a shadowed variable with the same name. Calling to\_lowercase on the query is necessary so no matter whether the user’s query is “rust”, “RUST”, “Rust”, or “rUsT”, we’ll treat the query as if it was “rust” and be insensitive to the case.

Note that query is now a String rather than a string slice, because calling to\_lowercase creates new data rather than referencing existing data. Say the query is “rUsT”, as an example: that string slice doesn’t contain a lowercase “u” or “t” for us to use, so we have to allocate a new String containing “rust”. When we pass query as an argument to the contains method now, we need to add an ampersand because the signature of contains is defined to take a string slice.

Next, we add a call to to\_lowercase on each line before we check whether it contains query to lowercase all characters. Now that we’ve converted line and query to lowercase, we’ll find matches no matter what the case of the query is.

Let’s see if this implementation passes the tests:

running 2 tests

test test::case\_insensitive ... ok

test test::case\_sensitive ... ok

test result: ok. 2 passed; 0 failed; 0 ignored; 0 measured

Great! They passed. Now, let’s call the new search\_case\_insensitive function from the run function. First, we’ll add a configuration option to switch between case-sensitive and case-insensitive search to the Config struct:

src/lib.rs

pub struct Config {

pub query: String,

pub filename: String,

pub case\_sensitive: bool,

}

Note that we added the case\_sensitive field that holds a Boolean. Next, we need the run function to check the case\_sensitive field’s value and use that to decide whether to call the search function or the search\_case\_insensitive function, as shown in Listing 12-22.

src/lib.rs

pub fn run(config: Config) -> Result<(), Box<Error>>{

let mut f = File::open(config.filename)?;

let mut contents = String::new();

f.read\_to\_string(&mut contents)?;

let results = if config.case\_sensitive {

search(&config.query, &contents)

} else {

search\_case\_insensitive(&config.query, &contents)

};

for line in results {

println!("{}", line);

}

Ok(())

}

Listing 12-22: Calling either search or search\_case\_insensitive based on the value in config.case\_sensitive

Finally, we need to check for the environment variable. The functions for working with environment variables are in the env module in the standard library, so we want to bring that module into scope with a use std::env; line at the top of src/lib.rs. Then we’ll use the var method from the env module to check for an environment variable named CASE\_INSENSITIVE, as shown in Listing 12-23.

src/lib.rs

use std::env;

// --snip--

impl Config {

pub fn new(args: &[String]) -> Result<Config, &'static str> {

if args.len() < 3 {

return Err("not enough arguments");

}

let query = args[1].clone();

let filename = args[2].clone();

let case\_sensitive = env::var("CASE\_INSENSITIVE").is\_err();

Ok(Config { query, filename, case\_sensitive })

}

}

Listing 12-23: Checking for an environment variable named CASE\_INSENSITIVE

Here, we create a new variable case\_sensitive. To set its value, we call the env::var function and pass it the name of the CASE\_INSENSITIVE environment variable. The env::var method returns a Result that will be the successful Ok variant that contains the value of the environment variable if the environment variable is set. It will return the Err variant if the environment variable is not set.

We’re using the is\_err method on the Result to check whether it’s an error and therefore unset, which means it should do a case-sensitive search. If the CASE\_INSENSITIVE environment variable is set to anything, is\_err will return false and will perform a case-insensitive search. We don’t care about the value of the environment variable, just whether it’s set or unset, so we’re checking is\_err rather than unwrap, expect, or any of the other methods we’ve seen on Result.

We pass the value in the case\_sensitive variable to the Config instance so the run function can read that value and decide whether to call search or search\_case\_insensitive as we implemented in Listing 12-22.

Let’s give it a try! First, we’ll run our program without the environment variable set and with the query “to”, which should match any line that contains the word “to” in all lowercase:

$ cargo run to poem.txt

Finished dev [unoptimized + debuginfo] target(s) in 0.0 secs

Running `target/debug/minigrep to poem.txt`

Are you nobody, too?

How dreary to be somebody!

Looks like that still works! Now, let’s run the program with CASE\_INSENSITIVE set to 1 but with the same query “to”; we should get lines that contain “to” that might have uppercase letters:

$ CASE\_INSENSITIVE=1 cargo run to poem.txt

Finished dev [unoptimized + debuginfo] target(s) in 0.0 secs

Running `target/debug/minigrep to poem.txt`

Are you nobody, too?

How dreary to be somebody!

To tell your name the livelong day

To an admiring bog!

Excellent, we also got lines containing “To”! Our minigrep program can now do case-insensitive searching controlled by an environment variable. Now you know how to manage options set using either command line arguments or environment variables!

Some programs allow arguments and environment variables for the same configuration. In those cases, the programs decide that one or the other takes precedence. For another exercise on your own, try controlling case insensitivity through either a command line argument or an environment variable. Decide whether the command line argument or the environment variable should take precedence if the program is run with one set to case sensitive and one set to case insensitive.

The std::env module contains many more useful features for dealing with environment variables: check out its documentation to see what is available.

Writing Error Messages to Standard Error Instead of Standard Output

At the moment we’re writing all of our output to the terminal using the println! function. Most terminals provide two kinds of output: standard output (stdout) for general information and standard error (stderr) for error messages. This distinction enables users to choose whether to direct the successful output of a program to a file but still print error messages to the screen.

The println! function is only capable of printing to standard output, so we have to use something else to print to standard error.

Checking Where Errors Are Written to

First, let’s observe how the content printed by minigrep is currently being written to standard output, including ant error messages we want to write to standard error instead. We’ll do that by redirecting the standard output stream to a file while also intentionally causing an error. We won’t redirect the standard error stream, so any content sent to standard error will continue to display on the screen.

Command line programs are expected to send error messages to the standard error stream so we can still see error messages on the screen even if we redirect the standard output stream to a file. Our program is not currently well-behaved: we’re about to see that it saves the error message output to a file instead!

The way to demonstrate this behavior is by running the program with > and the filename, output.txt, that we want to redirect the standard output stream to. We won’t pass any arguments, which should cause an error:

$ cargo run > output.txt

The > syntax tells the shell to write the contents of standard output to output.txt instead of the screen. We didn’t see the error message we were expecting printed on the screen, so that means it must have ended up in the file. This is what output.txt contains:

Problem parsing arguments: not enough arguments

Yup, our error message is being printed to standard output. It’s much more useful for error messages like this to be printed to standard error and have only data from a successful run end up in the file when we redirect standard output this way. We’ll change that.

Printing Errors to Standard Error

We’ll use the code in Listing 12-24 to change how error messages are printed. Because of the refactoring we did earlier in this chapter, all the code that prints error messages is in one function, in main. The standard library provides the eprintln! macro that prints to the standard error stream, so let’s change the two places we were calling println! to print errors to use eprintln! instead.

src/main.rs

fn main() {

let args: Vec<String> = env::args().collect();

let config = Config::new(&args).unwrap\_or\_else(|err| {

eprintln!("Problem parsing arguments: {}", err);

process::exit(1);

});

if let Err(e) = minigrep::run(config) {

eprintln!("Application error: {}", e);

process::exit(1);

}

}

Listing 12-24: Writing error messages to standard error instead of standard output using eprintln!

After changing println! to eprintln!, let’s run the program again in the same way, without any arguments and redirecting standard output with >:

$ cargo run > output.txt

Problem parsing arguments: not enough arguments

Now we see the error onscreen and output.txt contains nothing, which is the behavior we expect of command line programs.

Let’s run the program again with arguments that don’t cause an error but still redirect standard output to a file, like so:

$ cargo run to poem.txt > output.txt

We won’t see any output to the terminal, and output.txt will contain our results:

output.txt

Are you nobody, too?

How dreary to be somebody!

This demonstrates that we’re now using standard output for successful output and standard error for error output as appropriate.

Summary

In this chapter, we’ve recapped some of the major concepts you’ve learned so far and covered how to do common I/O operations in a Rust context. By using command line arguments, files, environment variables, and the eprintln! macro for printing errors, you’re now prepared to write command line applications. By using the concepts in previous chapters, your code will be well organized, store data effectively in the appropriate data structures, handle errors nicely, and be well tested.

Next, we’ll explore some functional-language-influenced Rust features: closures and iterators.